Discovery and Application of Network Information

Bruce Lowekamp

October 2000

CMU-CS-00-147

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Thomas Gross, Co-Chair
David O'Hallaron, Co-Chair
Peter Steenkiste
Francine Berman, University of California, San Diego

Effort sponsored by the Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

Keywords: Distributed systems, performance prediction, topology discovery, network performance, adaptive applications

Abstract

Distributed computing has brought about promising new possibilities, both by increasing the computing power to which people have access and by supporting new technologies such as real-time data analysis and collaborative applications. The power of distributed systems is offset by the tremendous complexity of developing applications for dynamic, heterogeneous environments. An important way to manage distributed applications is designing them to adapt their computing and networking needs to their environment. To support adaptation, a number of systems provide resource information obtained using active benchmarks. Benchmarks provide support for many applications, but their effectiveness is limited by low scalability, invasiveness, and the inability to derive network topology. I have examined the use of low-level network information to support adaptive applications without the shortcomings of active benchmarking.

The low-level details obtained directly from network components provide the information needed by distributed applications to adapt themselves to modern network environments. Low-level access overcomes the limitations inherent in benchmarking by providing a scalable, non-invasive measurement technique that provides network topology information while continuing to support the predictions of end-to-end application performance available through benchmarking. In this dissertation, I address the need for low-level information, the feasibility of providing it through an application-level interface, the accuracy of end-to-end predictions made provided by low-level information, and the topology discovery capabilities using low-level information. The topology discovery algorithms I present are the first to use the incomplete information available through network components and are provably good with minimal knowledge. My research demonstrates that violating the end-to-end networking abstraction by providing applications with access to low-level network knowledge meets the needs of many applications and is feasible on modern networks.

Acknowledgments

I cannot thank all of the people who have contributed in some way to the research that has become part of this dissertation. There are just too many friends and colleagues at CMU and elsewhere who have offered advice or help at different points. There are, however, a number of specific people whom I wish to thank.

Throughout my academic career, I have been lucky to have good advisors to guide my research process. At Virginia Tech, John Schug gave me my first experience with research as a freshman who knocked on his door one day. As my interests focused more on the computer science side of computational science, Layne Watson provided good advice for my undergraduate thesis research on simulating heat flow.

At CMU, I began working with Adam Beguelin, who supported my interests in exploring new ideas outside the original goals of the Dome project. When Adam left, Thomas Gross was a good match for my interests and offered support and insight as my earlier research led me to work with the Remulac team on the development of Remos. I have also been advised by Dave O'Hallaron, whose ability to step back, see the big picture, and ask the really important question has improved my skills tremendously.

As well as my advisors, I have worked with a number of excellent people in my research groups whose contributions are so inexorably bound to my dissertation that I could never hope to identify all of them. Although never my advisor, Peter Steenkiste has been involved in the research I've been doing since I came to CMU. The contributions of Mike Starkey, Matt Zekauskas, Nancy Miller, and Dean Sutherland have been important during my research. My long-time officemates, Marius Minea and Ralph Melton, made my time at CMU much easier. The time I have spent with Peter Dinda discussing research, statistics, and data gathering has been invaluable, and being able to use his research results to support my own research allowed me to focus more on the problems most interesting to me.

I am thankful to God for all that He has given me, and especially the people at Mount Hope Community Church, who have been an important part of my family while in Pittsburgh.

The support of my family and the lessons they taught me have been invaluable. The support, encouragement, and examples my Mom and Dad have given me throughout my life gave me the strength to accomplish what I have.

Most importantly, I am thankful to my wife, Jennifer, without whose love and support over the past ten years I would never have made it here.

Contents

Abstract iii

Acknowledgments iv

Contents v

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 How applications use the network . . . 3
1.1.1 Neighborhood communication . . . 4
1.1.2 Global communication . . . 4
1.2 About network congestion . . . 5
1.2.1 Congestion and end-to-end protocols . . . 6
1.2.2 Locations of congestion . . . 7
1.3 ECO . . . 9
1.3.1 Related work on collective communication . . . 9
1.3.2 ECO's network characterization . . . 10
1.3.3 ECO's communication pattern . . . 11
1.3.4 Example . . . 13
1.3.5 Micro-benchmarks . . . 13
1.3.6 Application programs . . . 13
1.3.7 Lessons from ECO . . . 16
1.4 Network measurement techniques . . . 16
1.4.1 Application-based . . . 17
1.4.2 Benchmark-based . . . 17
1.4.3 Packet train analysis . . . 18
1.4.4 Limitations of these techniques . . . 19
1.5 Network-based measurement . . . 22
1.6 Using network-based prediction . . . 23

2 Remos **26**
2.1 Overview . . . 26
2.2 Design challenges . . . 28
2.2.1 Dynamic behavior . . . 28
2.2.2 Sharing . . . 28
2.2.3 Information diversity . . . 29
2.2.4 Heterogeneity . . . 29
2.2.5 Level of abstraction . . . 30
2.3 Remos design principles . . . 31
2.4 Flow-based queries . . . 32
2.4.1 Multiple flow types . . . 32
2.4.2 Simultaneous queries and sharing . . . 33
2.4.3 Example . . . 33
2.5 Topology queries . . . 35
2.5.1 Examples . . . 37
2.6 Limitations . . . 39
2.7 Application programming interface . . . 39
2.7.1 Status functions . . . 39
2.7.2 Fitting functions . . . 40
2.7.3 Topology functions . . . 40
2.8 Using the API . . . 41
2.8.1 Adaptive applications . . . 41
2.8.2 Clustering . . . 42
2.9 Implementation . . . 44
2.10 Related work . . . 47
2.11 Summary . . . 48

3 Network-based Measurement **50**
3.1 Network requirements . . . 51
3.1.1 Requirements . . . 51
3.1.2 Networking technologies . . . 52
3.2 Network-based prediction . . . 54
3.2.1 Record a series of measurements . . . 54
3.2.2 Finding the bottleneck . . . 55
3.2.3 Time-series prediction . . . 55
3.2.4 Application mapping . . . 56
3.3 Testbed verification . . . 58
3.3.1 Experimental method . . . 59
3.4 Simulated verification . . . 63
3.4.1 NS simulator . . . 63
3.4.2 Packet traces . . . 63
3.4.3 Simulated network . . . 64
3.4.4 Results for heavily congested networks . . . 65
3.4.5 Results for moderate congestion . . . 67

3.5 Statistical metrics 67
3.6 Scalability 68
3.6.1 Scalability requirements 69
3.6.2 Benchmark scalability 73
3.6.3 Practical implementation 76
3.7 Related work 76
3.8 Conclusions 77

4 Topology Discovery 78
4.1 Network structure 78
4.1.1 IP routing 79
4.1.2 Bridged Ethernet 80
4.2 Effects of topology on applications 81
4.2.1 Benchmark-based topology 84
4.2.2 Motivation for explicit topology 85
4.3 Processor selection example 87
4.3.1 Node selection procedure 88
4.3.2 Experimental setup 90
4.3.3 Results 91
4.3.4 Related work 92
4.3.5 Lessons from cluster selection 92
4.4 Local gap example 93
4.4.1 Motivation for the local gap 93
4.4.2 Determining the network load 94
4.4.3 Heuristic for approximating the network load 95
4.4.4 Local gap 96
4.4.5 Simulated verification 96
4.4.6 Simulation results 99
4.4.7 Lessons from the local gap 100
4.5 Ethernet topology discovery algorithm 100
4.5.1 Implementation 102
4.5.2 Failures of the direct connection theorem 102
4.5.3 Related work 103
4.6 Topology discovery with incomplete knowledge 104
4.6.1 Rigorous presentation 108
4.6.2 Practicality 111
4.6.3 Specialization for traversal 113
4.7 Implementation 113
4.7.1 Preparation 113
4.7.2 Learning 114
4.7.3 Deriving the topology 115
4.7.4 Mapping algorithm 116
4.7.5 Virtual switches 117
4.7.6 Endpoint mapping 117

4.7.7 Cleaning up the topology . . . 118
4.8 Results . . . 118
4.9 Practical considerations . . . 120
4.10 Conclusions . . . 121

5 Conclusions **123**
5.1 Summary . . . 123
5.2 Contributions . . . 124
5.3 Future work . . . 125
5.4 Conclusion . . . 127

Bibliography **128**

List of Figures

1.1 Example of the topology of a typical Ethernet LAN 8
1.2 Algorithm used for partitioning the network into subnets 12
1.3 ECO's behavior on a heterogeneous network 14
1.4 Hierarchical structure of LANs and WANs 20
1.5 Four similar networks with differing performances 21
1.6 Conceptual diagram of performance prediction techniques 24

2.1 Overview of the Remos architecture 27
2.2 Query abstractions provided by Remos 31
2.3 Step-by-step example of a Remos flow query 34
2.4 How Remos represents routing, full-duplex, and half-duplex links 36
2.5 Remos graph representing the structure of a simple network 37
2.6 Representation of a testbed network 38
2.7 Detailed picture of the Remos architecture 46

3.1 The stages involved in making a network-based prediction 54
3.2 A path with correlated utilizations 55
3.3 Topology of the testbed used for experiments 59
3.4 The process used for analyzing the data collected in the experiment 60
3.5 Cumulative relative error distributions for the testbed experiment 62
3.6 Topology used in simulated experiments 64
3.7 Relative error for all measurements in simulation 66
3.8 Relative error for simulation experiment 68
3.9 Quantile-Quantile plot of CDFs for prediction techniques 69
3.10 A simple way to connect 16 machines with 100Mb Ethernet 73

4.1 Views of a network at the application, IP, and Ethernet layers 79
4.2 Performance predictions errors caused by missing bridges 80
4.3 Four similar networks with different performances 82
4.4 QuakeViz pipeline partitioned in different ways for specific environments . 86
4.5 Example of the concurrent status of frames in the QuakeViz pipeline 87
4.6 Node selection algorithm 89
4.7 Performance of FDTD application versus local gap 99
4.8 How contradictions can be used to determine connections between bridges . 104
4.9 Examples of valid and invalid connections between two bridges 105

4.10 Using contradictions to eliminate impossible connections 106
4.11 Example of forwarding sets for an Ethernet topology 107
4.12 Example of through sets for an Ethernet topology 107
4.13 An topologically indeterminate Ethernet network 109
4.14 The minimal information needed to determine an Ethernet's topology . . . 110
4.15 Examples of indeterminate and determinate networks 112
4.16 Topology of the CMU CS Department bridged Ethernet 119

List of Tables

1.1 Time, in seconds, for a 16000 byte broadcast on eight DEC Alphas 13
1.2 CHARMM communication time on a network of eight DEC Alphas 14
1.3 CHARMM time on a congested network of eight DEC Alphas 15
1.4 CHARMM communication time on a switched network of SGI INDYs . . . 15
1.5 Dome communication time 16

4.1 Performance of node selection algorithm in a congested environment 91

Chapter 1

Introduction

Parallel computing has been used for decades as the ultimate resource for problems in the forefront of science, engineering, and security. Many problems are best addressed by throwing massive computing power at them. As our understanding of how to build the massively parallel systems and algorithms used to solve these problems has improved, we have been able to address problems ranging from huge astronomical simulations to subatomic nuclear physics application.

While traditional massively parallel processing (MPP) systems have become well-understood, new parallel computing environments have emerged. Beginning with PVM [49], parallel computing has spread from its initial domain of high-cost custom-built machines to availability anywhere there are a few machines and a network to connect them. Initially the development of distributed computing was motivated more by the desire for cheaper computing resources. This goal has been achieved—originally working with only high-end workstations rather than traditional PCs, it is now possible to use commodity systems that cost little more than a home PC for state-of-the-art parallel processing.

Perhaps more importantly than saving money, however, distributed computing provides the power to solve problems unaddressable by previous computing paradigms. One of the major restrictions of the MPP computing model is that all input and output remain on that same machine, perhaps being transfered later from one filesystem to another. Distributed systems can solve problems that naturally span multiple locations. With distributed computing, it is possible to perform computation on data sets stored on remote sites. Real-time processing can be performed on the data from scientific or medical instruments as they are used. People from different locations can work collaboratively while viewing output of the same program. And, of course, multiple parallel computing resources can be brought together to solve those problems too large for even the biggest dedicated MPPs.

One of the enabling factors in the development of distributed computing has been the improvement of the programming interfaces that allow distributed applications to make use of their resources. Prior to the development of PVM, it was possible to write distributed applications using standard networking protocols, but the complexity of managing such communication directly made such applications rare. PVM changed this by offering a uniform API that allowed the application programmer to view the various machines involved in the computation as a single "virtual machine." The PVM interface made using a distributed system just as easy as programming an MPP. The development of MPI [80] implementations that support heterogeneous, distributed systems enabled more people to use these systems by allowing programmers to use the same code to run on everything from the fastest MPP to the slowest network of workstations. Shared-memory programming has been similarly improved with standardized interfaces such as pthreads [84].

While these developments have improved the capabilities of parallel programming, the environments used are now extremely diverse, both in terms of hardware and software. The hardware used to support parallel applications still includes traditional MPPs. In addition, dedicated clusters of PCs, such as Beowulf [10] systems, may be used, offering widely varying levels of computing and networking support. Furthermore, some applications require access to resources spread across the Internet, with its high latencies and bandwidths varying from almost unlimited to modem speeds.

Applications, meanwhile, span a similar breadth. Some applications are programmed with a completely regular communication pattern, such as communication along a grid. Other applications have dramatically different communication patterns, ranging from master/worker applications where one processor dispatches tasks to the other processors to pipelined applications where processors are arranged in a pipeline and work is passed through them in sequence. Some applications, such as visualization, have real-time deadlines, requiring the application to acquire more resources or adjust the computations being performed to deliver on-time results to the user. Other applications, such as many simulations, have no such constraints, and the user may be most interested in optimizing the cost-effectiveness of the computation.

Despite the breadth of applications, the improvements in programming interfaces has allowed many of these applications to be written with the same set of programming tools. The abstractions provided to the modern programmer allow great power to be harnessed without any knowledge of how each particular parallel system will be utilized or how they perform.

While interfaces with better abstraction have improved the programmability of parallel systems, they have not improved the ability of a programmer to utilize these systems efficiently. Prior to the advent of standardized programming interfaces, it was typically necessary to customize the code for a particular machine. Optimization for a particular architecture was, therefore, much more convenient.

Beowulfs are an excellent example of why performance information is important. While MPP network architecture is generally fixed for a particular platform, a Beowulf's network can vary considerably, even though all Beowulf nodes are indistinguishable from the application's view. Using commodity Ethernet switches, it is possible to build a network with sufficient bandwidth to emulate a crossbar, or to create a bottleneck such that the bisection bandwidth of the network is no greater than the bandwidth of the endpoint links.

To solve these problems, an application must have the ability to characterize the network environment it will be using. By obtaining that information in a portable form, the application can adapt to its networking environment without losing the portability obtained by the high level of abstraction available in modern distributed programming environments.

Although in many cases there is only one particular portion of the network that causes bottlenecks, frequently the Internet in wide-area distributed applications, it is important that network performance information be available for all components of a distributed environment. The programming interface has allowed application developers to treat processors in an MPP, machines in a cluster, and machines across the world using the same interface. If portable selection of the most appropriate environment and optimization of behavior is the goal of providing network information, then it must also be available regardless of the types of machines involved. Only then will a developer be able to design an application without regard for its communication requirements and expect it to automatically select and configure itself for the proper environment without the application's end-user understanding the issues behind the adaptation.

1.1 How applications use the network

To understand the information that must be provided to distributed applications, we must first understand how applications make use of the network. Broadly speaking, applications' communication patterns are classified into two groups: regular and irregular. Regular applications have the same communication occurring between all processors. Irregular applications may use a number of different types of communication patterns. This section primarily discusses regular communication, because several of the following sections discuss how to optimize the communication requirements of regular applications.

Regardless of whether an application is regular or irregular, there are a number of different techniques it can use to adapt to the network. A very large number of projects have focused on various aspects of developing and supporting adaptive applications [12, 16, 27, 45, 47, 52, 86, 96, 100, 110, 115]. Steenkiste has divided the adaptation used by network-aware applications into three strategies: model-based, performance-based, and feature-based [106]. Of these three, I will focus on model-based adaptation, where an application predicts its performance using a model of its performance based on network metrics. Model-based adaptation allows a program to predict its performance on a large number of systems without actually running on them. While they frequently offer better performance, performance-based and feature-based adaptation are more limited in value because they only allow for adaptation on the machines already in use. Many applications may desire to use model-based adaptation prior to running or for dynamic load-balancing, while also using performance- or feature-based prediction during execution to adapt to the chosen set of resources.

One of the problems addressed by many applications in distributed environments is mapping a regular application onto a complex irregular network. Most previous mapping solutions have not attempted this problem, and have instead either addressed only average network performance given by a uniform system [113] or ignored the issue altogether. Some work [120] has modeled communication costs directly with brute-force calculations, but such approaches are difficult to use for processor selection due to their high cost.

Broad classifications of the communication needs of an application are possible, however, and can significantly benefit a scheduler. The most important distinction to make is between local and global communication. Most applications have communication which can be divided into one of these categories—some use both at different times. Local communication occurs along the application's structure, where each task exchanges messages with a small set of neighbors. Global communication involves all or most tasks in a single conceptual operation, where data is exchanged in some way between all tasks. This difference becomes very important with distributed systems that may have poor bisection bandwidth, but high local bandwidth. Some applications may exhibit both, such as FFT, which has two phases, one involving only neighborhood communication along rows or columns and the other involving a matrix transpose, generally done through all-to-all pairs communication.

1.1.1 Neighborhood communication

Neighborhood, or local, communication refers to communication between a task and the set of tasks that are neighbors to that task. The neighbors of a task are usually determined by the structure used by the application. For instance, in an application that arranges tasks along a ring, each task has two neighbors. In an application where tasks are arranged along a mesh, each task has four neighbors. A three dimensional irregularly partitioned structure, such as that used by Quake [8], may have variable numbers of neighbors depending on the particular partitioning of the space, however, even for such an irregular application, processors communicate with a restricted number of neighbors [88]. While the partitioning is irregular, the presence of neighborhood communication still offers opportunities for optimization.

The importance of separating global and local communication is that distributed networks are frequently better optimized for one than the other. For instance, in a 16 processor cluster where all processors are connected to a single switch with 100Mb Ethernet, modern processors can easily saturate each link. A high-performance switch can handle all links transmitting at full speed, allowing any global or local communication to occur at full link speed. On the other hand, if the same 16 nodes are connected with Ethernet set up in a binary tree with the same 100Mb links, the higher levels of the tree will quickly become bottlenecks for global communication, but local communication can occur at a higher aggregate rate if communication along the bottleneck links is minimized. The wide variety of network technologies now available, such as 100Mb and 1Gb Ethernet, ATM, and Myrinet, allow networks to be designed with varying performance characteristics and bottlenecks. Separating local and global communication allows networks and applications to be evaluated in terms of how they support and utilize each type of communication, thus simplifying the problems of performance prediction and node selection.

1.1.2 Global communication

Broadcasts, all-to-all pairs communication, and operations such as prefix sums are examples of global communication. The performance of global communication is generally dependent on the bisection bandwidth [42], which is the minimum communication bandwidth between any bisection of the processors. Calculating the exact bandwidth used by an application for global communication may be complicated by the presence of various hardware or software optimizations for that communication. Assuming sufficient knowledge of the system's behavior, the utilization of the bottleneck links in the network can be calculated. Knowledge of network hardware support for global communication relieves the scheduler of determining the best way to perform collective communication on that network.

There are a number of user-level and system-level possibilities for optimizing global communication. ECO, which is described in the next section, is a user-level system that arranges collective operations along a tree structure designed to reflect the local network's topology [73]. Magpie is a more recent system designed using similar techniques to take advantage of knowledge of WAN topology [67, 68]. Better performance can usually be achieved by taking advantage of network-level operations [56], but because these are not generally available, packages such as ECO and Magpie are needed for efficient performance on heterogeneous networks.

1.2 About network congestion

Before discussing methods for measuring and predicting congestion and performance, I will characterize network congestion, how it affects the applications, and expectations for change in the future.

In a network without any other traffic, the application's behavior is determined solely by the components of the network. The latency of the components along the data's path is additive, and the bandwidth obtained by the application is determined by the component on the path with the lowest bandwidth, referred to as the bottleneck bandwidth. For an application making a connection across such a network, this information is sufficient to determine the performance of that connection.

However, real applications seldom run on a network with no other traffic. In most environments, a large number of applications compete for use of the same bandwidth. As the amount of traffic sent across the network approaches its capacity, the performance obtained by the applications is reduced. There are two ways that traffic congestion can reduce performance.

Link congestion

The most elementary form of congestion is link congestion, or competition for the data link connecting the machine to the network. The majority of network architectures send data across a single serial line that cannot be used simultaneously by multiple hosts. In some network architectures (full-duplex point-to-point) there are no other machines sharing the line, so there is never link contention. Other architectures have several machines sharing the same link, and a machine may be competing with other senders when putting data on the link.

The best, and most common, example of a shared network is Ethernet [81]. The Ethernet protocol implements carrier sense, multiple access, and collision detection. Simply put, this means that several machines connected to the same shared Ethernet can detect if another machine is using the network and can react if two begin using it simultaneously.

This technique allows the machines to effectively share the same link. However, when several machines are trying to send data simultaneously, the contention reduces the bandwidth each receives.

Thinwire and thickwire Ethernets are obviously shared, however, sharing the same physical link is not the only way a data link can be shared. Twisted pair wiring appears to connect hosts directly to the network without sharing by other hosts. However, many hubs that these machines are connected to are merely copying the data between lines. While the hub provides separate physical, or electrical, links on each port, copying the data between ports results in the physical links acting as the same data link. Thus, the contention issues are identical to that of an electrically shared Ethernet.

Switch congestion

Ethernet switches, or bridges, on the other hand, actually forward the traffic between their ports based on the destination of the data. For switched Ethernet and many other types of networking, two machines sending data simultaneously do not compete for the same network link. Instead, competition occurs inside the switch. For instance, if the path taken by data sent by two different machines reaches the same switch, to that point there has been no competition. However, if the paths from that switch to the data's destinations leave the switch on the same link, then there is competition inside the switch for that outgoing link.

The exact policy by which this congestion is resolved varies from device to device and has a significant impact on the performance of the network. When data is arriving faster than it can leave, the switch may initially save some of that data in a buffer. If the higher rate is merely temporary, then the buffer may absorb a small burst and all data will be delivered as the buffer empties. However, if the data continues to arrive at a faster rate than the outgoing link, the buffer will fill up and some packets will have to be dropped. This behavior is commonly modeled as an M/1/1/m queue, although the arrival process is actually not Poisson.

Although there are some differences in the mechanisms behind link and switch congestion, the overall effect is the same. The latency required to deliver a packet of data increases and the bandwidth available for the application decreases. It is theoretically possible for a host to observe the link congestion present on its own interface, but this is rarely information provided to the user, so for most intents and purposes, there is no difference between the two as far as application behavior.

1.2.1 Congestion and end-to-end protocols

Mild congestion causes packets to be delayed in the network. Heavier congestion may cause switches to drop the packets. The important question is what effect delayed or dropped packets may have on the end-to-end performance of the connection. The answer to this question depends on the particular protocol being used.

Some data streams, and the protocols used to support them, are not reliable and may not care about packet loss. Audio or video signals frequently tolerate small numbers of lost packets. In the event of high packet loss, the amount of data being sent may be adjusted in hope of reducing the congestion so a higher percentage of packets are delivered successfully.

The vast majority of applications, however, need lossless network connections. Almost all of these applications rely on TCP to ensure in-order end-to-end delivery. Complete details of TCP's congestion avoidance strategies can be found in most networking textbooks, but here is a quick overview.

TCP assumes that all packet loss is due to congestion. On a modern wired network, this is a reasonable assumption. Whenever a packet is lost, TCP reduces its window size, which governs the amount of data in transit on the network simultaneously. This, in turn, restricts the rate at which TCP can send data. Because new packets are sent and the window enlarged only when previous packets are acknowledged, this behavior also adjusts the transmission rate for a connection's latency.

There are several variations of the basic TCP algorithm available. Because they recover from packet loss in different ways, their performance varies as they encounter congestion [17].

1.2.2 Locations of congestion

To understand how congestion affects traffic, first consider where congestion occurs in networks.

Local area networks

Almost all LANs are organized with a tree (or star) topology. Ethernet, the most common type of LAN, requires such a topology. Other networks are not so restrictive, but are frequently used in a tree topology as well.

The tree structure naturally creates bottlenecks. Consider the sample network shown in Figure 1.1. The switches in this network prevent link congestion between desktops A and B, although other machines attached to the same hubs may cause link congestion. Connections between A and B and many others in the network will experience congestion further along their paths. If A and B both try to connect to machines across the Internet, their connections will share the same links from the first switch out to the Internet connection. Furthermore, the relationship between the structure of the network and the building's structure can cause bottlenecks. For efficiency reasons, all of the servers for a department are typically located in a central machine room. In this network example, there is only a single link connecting all of the desktops in the network to the machine room, therefore all of the machines in the network will face congestion whenever they request a file or perform another operation involving the servers.

Effective design of Ethernet networks is a topic beyond the scope of this dissertation. The important point is that congestion in Ethernet networks is extremely dependent on the relationship between the topology of the network and placement of machines that are in communication with each other. Without an understanding of the topology of the network and the traffic sources on it, it is impossible to predict where congestion will occur, and correspondingly, to predict the performance of a new application on the network.

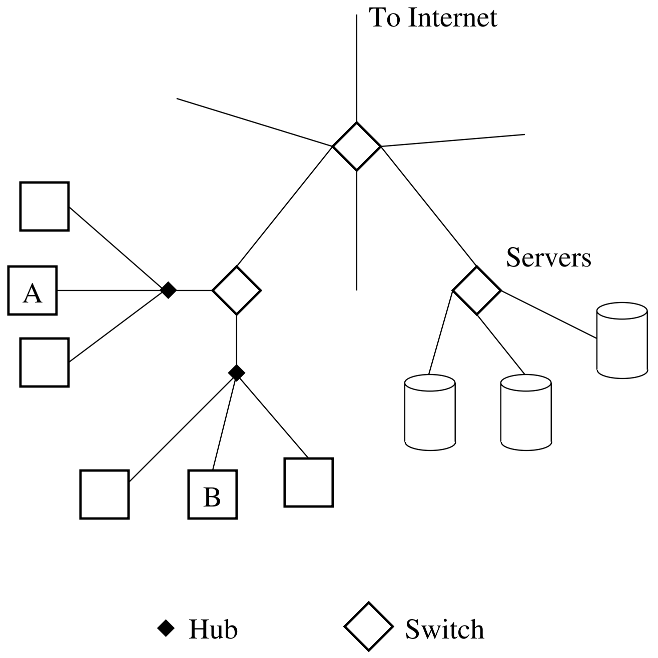


Figure 1.1: Example of the topology of a typical Ethernet LAN.

Wide area networks

Wide area networks are not restricted to the tree topology that local area networks are. In fact, the topology and actual routing of packets may be arbitrary: depending on contracts, pricing, and other issues independent of the capabilities of the networks. Furthermore, it is fairly dynamic: as links go up and down, capacity is transferred between voice and data, and traffic demands change.

In this environment, congestion can happen anywhere, and the network conditions causing the congestion can change at any time. Wide area ISPs may provide a specific bandwidth connection from their clients to their internal networks, but their internal bottlenecks and the bottlenecks of other networks needed to reach the destinations will frequently significantly reduce the bandwidth actually available for connections.

There are fundamental differences between LAN traffic and WAN traffic. The most important difference is the number of active connections on each network. LANs typically support few connections simultaneously. With so few connections, a new application can have a significant impact on the network's status and on other applications. A WAN, however, generally supports thousands of connections simultaneously. Because the competing traffic is an aggregation of many separate connections, an individual application can generally not influence the behavior of the other applications significantly.

The difference in traffic types on the network has a substantial influence on the amount of congestion generally seen. Because LAN traffic consists of only a few simultaneous connections, it is much more prone to burstiness, with applications occasionally waking up and briefly consuming the majority of the bandwidth, then sleeping. This makes the average congestion level rather low, but with bursts of maximum utilization. Because of the aggregation of traffic on WANs, however, their traffic tends to be completely different. No single application consumes a significant fraction of the bandwidth, so although the traffic is still bursty, it is much more predictable than LAN traffic. Furthermore, the congestion level is typically high. Because so many applications are sharing the network, they are capable of utilizing far more bandwidth than is typically available on a link.

1.3 ECO

This section gives one example of how an application's communication can be adapted to fit the network on which it is running. ECO is a good example of the types of measurements and adaptations that may be needed in a distributed environment.

One of the largest differences between distributed systems and dedicated MPPs is the lack of support for collective communication on distributed systems. Not only is there not native support, but the low bisection bandwidth severely restrains the performance of collective communication run at application level across the network. The Efficient Collective Operations package (ECO) contains programs which solve this problem by analyzing the network and establishing efficient communication patterns. These patterns are then used by ECO's collective operations, which are available to applications through a run-time library, as well as for task placement to improve the performance of local communication. The analysis to build the patterns is done off-line, so that, after paying the one-time cost of analyzing the network, the execution of application programs is not delayed.

This section describes ECO and gives performance results of using ECO to implement the collective communication in CHARMM, a widely used macromolecular dynamics package. The techniques used by ECO to measure the network's performance and to design the communication patterns are of particular interest. ECO was designed and implemented several years ago without any support from the network for measurement or optimization. It utilizes benchmarks that send data to measure the performance of the network. The performance information is then used to determine optimized communication patterns.

The lessens learned from developing ECO were among the major influences leading up to the design of Remos. In fact, the difficulties I experienced in determining the topology of the network needed for the communication patterns were among the fundamental influences behind my support of the topology interface in Remos. Because of these lessons, the Remos interface provides the information needed to meet all of ECO's topology knowledge requirements.

1.3.1 Related work on collective communication

Collective communication provides important functionality for many applications. Efficient implementations of core collective operations is crucial for achieving maximum performance of applications on message-passing systems [83].

There has been substantial work on collective communication packages. The MPI standard [46] acknowledges the importance of collective communication by including it as a chapter in the specification. Papers from the InterCom project discuss general techniques for building high-performance collective communication libraries, implementation of their library on several architectures [9], and other packages and approaches to collective communication [83].

Bala et al. describe a collective communication library originally designed for the IBM SP1 [6]. They discuss performance tuning issues and provide a detailed discussion of the semantics of collective communication and group membership, including the correctness of collective operations.

Considerable work has been done on collective operations and multicast communication in general, which can be used as the underpinnings for collective communication, on a variety of specific physical networks. McKinley et al. have written several papers on issues involved in implementing such operations on bus-based networks [78], wormhole routed MPPs [79], and ATM networks [57].

Many applications require the notion of communication topology, such as a mesh or a ring, which is used by the application for nearest neighbor communication. Developers have expressed a desire for a basic set of topologies [105] for both programming ease and efficiency reasons. PVM currently does not support topologies. The MPI standard dedicates a chapter to the discussion of a mechanism for describing arbitrary topology needs to the message passing system [46].

Of particular interest to ECO development is research done on grouping hosts on the basis of network topology. This technique has been used in two areas. Evans and Butt make use of this technique to facilitate load balancing [43]. In their approach, full load balancing information and communication occurs within subnets, while communication between subnets is more carefully controlled. Also related to this subject is the work of Efe on grouping related tasks together in a subnet for a system with deterministic task dependencies [41]. Both of these systems share ECO's principle of limiting communication between subnets. However, a major difference between these systems and ECO is that they attempt to avoid global communication whereas ECO tries to optimize it. Furthermore, only ECO addresses the issue of automatically identifying subnets. Magpie [67, 68] was developed after ECO and performs similar optimizations as ECO, focusing instead on optimizing MPI-based collective communication across wide area networks.

1.3.2 ECO's network characterization

Several techniques were examined in the hope of developing a portable technique for automatically and robustly adapting communication patterns to network topology. ECO uses a simple benchmark where the communication time between two hosts is measured by timing round-trip messages sent using PVM. This metric is desirable because it expresses the sum of all these terms, including the computational overhead of the message passing library, in a single measurable quantity.

To measure these times, a program is run on all hosts that are anticipated to be used for running parallel programs. A host exchanges a message with another host several times. The total time is divided by twice the number of round trips and recorded. Several round trips are used for each measurement to minimize the influence of clock granularity. The communication time is measured with one pair of hosts exchanging messages at a time, in order to prevent the program from causing network congestion that would distort the results. This exchange is repeated for each pair of hosts, and the whole process is repeated several times. This technique assumes a single route between machines.

Although this measurement process is an $O(p^2)$ process when run on p machines, for the environments for which ECO was designed, p was small, so it did not take excessively long and required little processor time, which should make it inexpensive for those who pay for CPU time. Since these results are used to determine physical network topology and are saved to disk, it is only necessary to perform the characterization on occasions when a change is made in the network or the location of machines.

After the measurements are completed, it is necessary to label the communication time for each pair of hosts with a single value. It seems intuitive that the mean observed time would most accurately reflect the cost to communicate between hosts. However, experience has shown that the mean time is a poor indicator, due to the large delay that can be caused by collisions. The high cost of delays caused by a brief burst of heavy traffic can be made worse by the exponential back-off scheme used on Ethernet. A single exchange that experiences this type of effect can skew the mean so that hosts that may share the same network bus appear to have a worse network connection than those hundreds of miles away. Although these collisions result from ambient traffic and should be accounted for, it would be necessary to run the analysis over a period of several hours to gain even an approximate measure of the frequency of these occurrences.

Other possible measures are the minimum, maximum, and median times. The maximum time will reflect the worst collision that occurred, as discussed above. The median time is likely not to reflect the delays caused by infrequent collisions or traffic burstiness if enough measurements are taken. The minimum time has been chosen for use by ECO, because ECO uses these measurements to determine the physical structure of the network, and the minimum time most accurately reflects this, as it reflects the least influence from ambient traffic. Using the minimum time also has the advantage of allowing the smallest number of measurements to obtain an accurate result. These results, which characterize the throughput of the communication links, are stored for use by the partitioning algorithm.

1.3.3 ECO's communication pattern

ECO begins by sending data between every pair of machines in the network, building the complete matrix of communication performance. It uses this information to partition the machines in the network into clusters of machines. Then a communication pattern is built between the clusters.

The steps involved in building the pattern are:

1. Divide the overall network into "subnets" consisting of hosts in the same physical network. A host is on the same subnet as other hosts with which it has its lowest edge costs.

2. Position the originator of the broadcast at the root of the tree.

```
initialize subnets to empty

for all nodes
    node.min_edge = minimum cost edge incident on node

sort edges by nondecreasing cost

for all edges(a,b)
    if a and b are in the same subnet
       continue
    if edge.weight > 1.20 * node(a).min_edge or
       edge.weight > 1.20 * node(b).min_edge
       continue
    if node(a) in a subnet
       if (edge.weight > 1.20 * node(a).subnet_min_edge)
         continue
    if node(b) in a subnet
       if (edge.weight > 1.20 * node(b).subnet_min_edge)
         continue
    merge node(a).subnet and node(b).subnet
    set subnet_min_edge to min(edge, node(a).subnet_min_edge,
         node(b).subnet_min_edge)
```

Figure 1.2: Algorithm used for partitioning the network into subnets

3. Create a tree using the subnets as vertices rather than the individual processors. Edges on the tree now represent inter-subnet communication.

4. Optimize the intra-subnet communication using patterns appropriate to each subnet's network type.

The algorithm used to partition hosts into subnets is given in Figure 1.2. The key criteria for subnet membership is that the cost of the edge between them be within 20% of the cost of the least expensive edge incident to each of them and within 20% of the least expensive edge within the subnet. Although arbitrary, the 20% cutoff has proven excellent at accurately partitioning networks of machines available at CMU and PSC.

Note than when a node is added to a subnet the algorithm does not check that the cost of all edges from the new node to members of that subnet are within 20% of the new node's minimum cost edge. This has the advantage that a small number of inaccurate measurements can be made in the network timings without causing incorrect partitioning. This fault tolerance allows for the initial measurements to be taken more quickly, since a single inaccurate measurement should have few consequences.

The partitioning need only be done once per network. Characterization and partitioning need only be redone by the user when the physical network changes. The results of the partitioning are stored in a file which is loaded when an ECO program is run.

method	mean(std.dev.)	median	minimum
ECO	0.119(.008)	0.0651	0.0614
tree($k = 2$)	0.174(.007)	0.128	0.104
tree($k = 3$)	0.162(.007)	0.122	0.104
star	0.141(.007)	0.109	0.104

Table 1.1: Time, in seconds, for a 16000 byte broadcast on eight DEC Alphas

1.3.4 Example

Figure 1.3 shows the application of ECO's technique to a network of machines at CMU and PSC. Figure 1.3(b) shows that ECO successfully distinguishes between networks with large differences in performance, such as switched FDDI and Ethernet, as well as between Ethernet networks separated by a bridge. The ring-topology generated by ECO is shown in Figure 1.3(c). The broadcast tree generated by ECO is shown in Figure 1.3(d).

1.3.5 Micro-benchmarks

Due to the inherent difficulty in timing a parallel operation, only the performance of operations that are started from the root, such as broadcast, have been measured directly. The measurements were taken by determining the offsets between system clocks among the machines used, recording the time before the operation began, and determining the latest time at which the message was received. Clock drift or adjustments can be a factor in such experiments, but drift was not a factor over the time period of the measurements, and the machines were not running a daemon, such as xntpd, that would adjust their clocks during execution.

Results are shown in Table 1.1 for the broadcast of a message of 16000 bytes on eight DEC Alphas, distributed among three 10 Mb Ethernet segments joined by a Cisco 7505 bridge. In this case, the butterfly pattern would produce the same results as the binary tree. The measurement was repeated 1000 times with relatively little ambient traffic.

1.3.6 Application programs

ECO has been used to provide collective communication for CHARMM [21], a macromolecular dynamics program used by many chemists. CHARMM's implementation of collective communication uses a butterfly pattern, the performance of which has been optimized fairly heavily. It uses PVM with in-place data packing and has almost no computational overhead in its collective communication routines. The collective communication used by CHARMM in this benchmark consists of a large number of broadcasts, gather-to-all, and reduce-to-all operations.

The same set of eight DEC Alphas used in Section 1.3.5 was used to run CHARMM. The measurements were taken during periods of low ambient traffic and repeated five times. Table 1.2 shows the results.

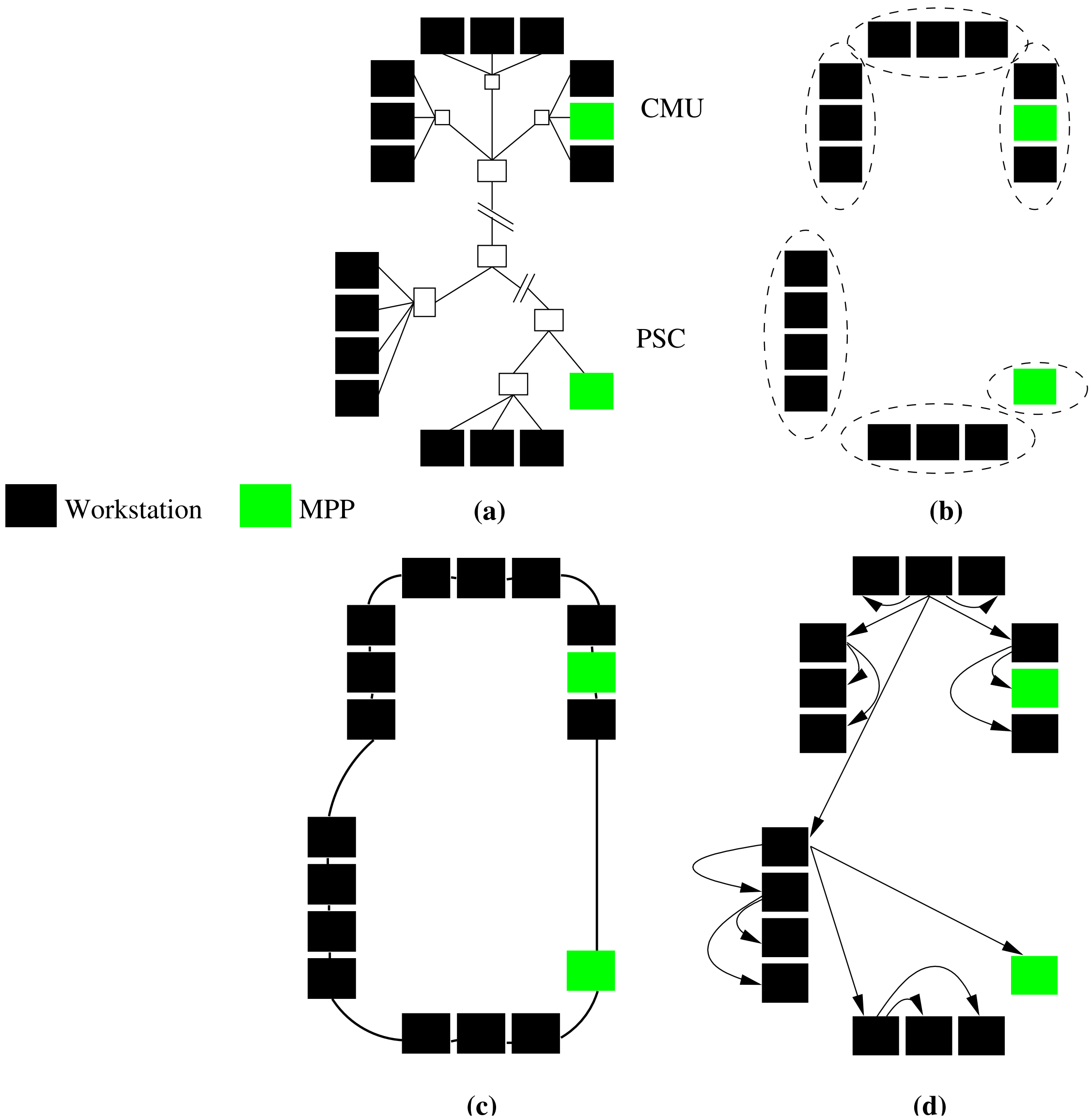


Figure 1.3: ECO's behavior on a heterogeneous network: a) physical network, b) ECO's partitioning into subnets, c) ECO derived ring topology, and d) ECO derived collective communication pattern

pattern	mean(std.dev)	median	minimum
ECO	2.1(.2)	2.17	1.79
tree($k = 2$)	2.7(.1)	2.66	2.65
tree($k = 3$)	2.3(.1)	2.30	2.12
star	2.7(.2)	2.59	2.47
CHARMM butterfly	3.0(.4)	2.95	2.69

Table 1.2: Communication time, in minutes, for CHARMM running on a network of eight DEC Alphas

pattern	mean
ECO	13.9
tree($k = 2$)	18.7
tree($k = 3$)	21.6
CHARMM butterfly	16.6

Table 1.3: Communication time, in minutes, for CHARMM running on a network of eight DEC Alphas with high ambient traffic

pattern	time
ECO($k = 2$)	1.08
ECO($k = 3$)	3.28
CHARMM butterfly	0.82

Table 1.4: Communication time, in minutes, for CHARMM on a switched Ethernet network of eight SGI INDYs

Measurements were also taken on an Ethernet with high levels of ambient traffic. These are shown in Table 1.3. In order to minimize intrusion upon other users in our environment, only two runs were made. Therefore, additional statistical results are not presented.

In order to evaluate the overhead of ECO's routines, CHARMM was also run on a set of eight SGI INDYs connected with switched 10 Mb Ethernet. This type of network should be ideal for the butterfly patterns used by CHARMM's native communication. ECO was run with $k = 2$ and $k = 3$ trees for the single subnet. (Note that since there is only one subnet, choice of the local pattern is the only issue to be considered.) There was no other traffic on the network. The results in Table 1.4 indicate that there is a slight overhead inherent in using ECO, as indicated by the results for $k = 2$. The results for $k = 3$ show the importance of matching the appropriate communication pattern to each subnet. Using wider trees is important on bus-based networks, since it reduces the number of messages that are attempted in parallel, but it degrades the performance on a switched network that can handle the aggregate bandwidth.

ECO has also been used to implement the collective communication for Dome [3]. A molecular dynamics program written in Dome has been run on a network of 20 machines, consisting of six DEC Alphas attached to two Ethernets, five IBM Power PCs attached to an Ethernet, two DEC Alphas attached to switched FDDI, and seven SGI INDYs attached to switched Ethernet. These tests were run using four communication patterns: ECO's optimized pattern, star, tree ($k = 4$), and ring. The times for communication required by this application are shown in Table 1.5. I have not yet run CHARMM on a heterogeneous collection of machines.

Dome also makes use of ECO's topology function, using a ring topology for its load-balancing communications. Use of this function prevents the user from having to order the machines by hand to ensure quick load-balancing and reduces the load-balancing communication time by more than half compared to times for a randomly distributed arrangement of nodes.

pattern	mean
ECO	63.0
star	153.5
tree($k = 4$)	148.1
ring	414.2

Table 1.5: Communication time, in seconds, for a Dome molecular dynamics application

1.3.7 Lessons from ECO

The algorithm in Figure 1.2 presented several problems at the time of development and as time passed. While developing the code, it was difficult to obtain measurements that reliably and consistently differentiated between the different portions of the network. Although those problems were mostly solved by careful implementation, it remained necessary to check the output of the network partitioning code for any obvious mistakes caused by unusual network behavior during the analysis.

Furthermore, as the networks at CMU and PSC that ECO was originally developed on were upgraded, it became even more difficult to distinguish between the different portions of the network using such simple tools. Although there were still bottlenecks between different segments of the network, the switching latencies quickly decreased. If the bottleneck segments were not congested during the measurements, then there were sometimes no measurements to indicate that a bottleneck segment existed and would cause a problem when the application attempted several simultaneous data transfers across that segment.

Even when done accurately, the time required to perform the $O(P^2)$ measurements was quite high. This was acceptable for a package that was intended to be run once to capture the physical topology of the network and the results saved, but inappropriate if more current information was desired. More importantly, if more machines were added to the environment, the entire experiment had to be repeated.

The lessons from ECO were not, however, entirely negative. Once the data was collected, the performance gains achieved using the information in a rather straightforward manner were significant. These performance improvements were what led me to pursue a better way to obtain network topology and utilization information.

1.4 Network measurement techniques

ECO is an example of using one type of benchmarking to measure the performance of a network. This section discusses benchmarking, as well as other techniques used to measure the performance of a network.

Understanding the types and causes of network congestion is important for designing techniques to measure and predict that congestion. As described in Section 1.2, network congestion

- can occur in all types of networks,

- is due to competing traffic from other sources, which is constantly changing,
- is dependent on the networking architecture and dropping policies of the hardware and software, and
- is dependent on the topology of the network, which governs the path needed to go from source to destination and the competing traffic met along the way.

Given these complexities, this section focuses on several different techniques for discovering the behavior of the network.

All the bandwidth prediction techniques described here rely on the same basic time series prediction models, which use a series of measurements to make predictions of future behavior. The difference between the three techniques is what measurements are taken and how they are converted to a prediction of application performance. Ideally, the series of measurements is taken by an independent daemon that collects the data for later use when a user wishes to run an application. A mathematical model is fit to the series. When a user requests a prediction, future performance is extrapolated from the model that has been fit to the past data. Selection and use of time series models has been dealt with by many authors [19, 36, 116]. In my notation, time series models are indicated by a t subscript on the measurement that is used for the series.

Regardless of the model chosen for prediction of future behavior, the first choice is selecting a method for determining the network's current status. Once that information is discovered, a variety of techniques can be chosen to make the actual prediction.

1.4.1 Application-based

The most straightforward measure of an application's performance is obtained by running the application on the network. Similarly, the most straightforward prediction of an application's future performance on that network is obtained using the application's performance history on that network to predict its future performance. The performance of an application A running on a network $\mathcal{N}$ is denoted $A(\mathcal{N})$. The time series model $A_t(\mathcal{N})$ can be used to predict the application's performance on the network.

Unfortunately, the many combinations of applications, parameters, and resource selections make gathering enough application history information to provide useful predictions infeasible. For this reason, other prediction techniques must be considered.

1.4.2 Benchmark-based

Benchmarking solves many of these problems by using a small set of representative applications, called benchmarks or probes, to predict the performance of many applications. The performance of the benchmarking application is denoted $B(\mathcal{N})$. Again, a time series of benchmarks can be used to form a prediction, $B_t(\mathcal{N})$, of how the benchmark B will perform on the network $\mathcal{N}$ in the future.

The challenge with using benchmarks for performance predictions is the mapping from benchmark performance to application performance. One method is to make the assumption that the relative performance of the application and benchmarks are the same, so the best connection for the application is assumed to be the same as the best connection for the benchmark. This approach is often useful for parallel applications whose only concern is moving the data as quickly as possible.

The lack of quantitative information about the application's performance, however, prevents this technique from being useful in many situations. It does not answer the question of which connections are sufficient for the application's needs, nor does it provide information necessary for setting application quality parameters. Quantitative information is needed for these decisions.

Benchmarking can be used to provide quantitative information, and for some applications, a benchmark will perform similar operations so that the results can be used with a simple rescaling. In other cases, such as using a TCP-based benchmark to predict the performance of a multimedia application that can handle loss, one must develop a model of the network conditions that caused the benchmark's performance and then determine how the application will perform under those same conditions. A mapping function converting the predicted performance of the benchmark to a model of the network can be written as $M(B_t(\mathcal{N}))$. A prediction of application performance based on this network model can be written $A_n(M(B_t(\mathcal{N})))$, where the subscript n is used to denote a performance prediction for the application A based on a network resource model.

1.4.3 Packet train analysis

Rather than using end-to-end applications to benchmark network performance, it is also possible to derive network performance from more fundamental end-to-end behaviors of a network. Packet train analysis is one technique of doing this. Rather than using actual application data, packet train analysis involves sending a series of packets back-to-back, referred to as a packet train. By observing the interarrival time of these packets, which is determined by the congestion encountered along the path, the bandwidth available on the network can be inferred.

Carter and Crovella's packet-train probes, called bprobe and cprobe, use interarrival times of ping packets to measure the bottleneck and available bandwidths along a path [24]. Bprobe measures the minimum interarrival time of a series of pings. With the assumption that the minimum time corresponds to successfully queueing two packets in sequence on the bottleneck link, the time, when measured accurately, reflects the bandwidth of that link. This technique implies a number of assumptions about the networking hardware and software, such as FCFS tail-drop queueing and symmetric routes. Cprobe measures the time a series of ping packets take to make the round trip, and is used to determine the competing traffic present while the packets are being transmitted.

A packet train approach is an example of a minimally invasive active measurement. Although it can learn more about the network by sending less information than a pure application-level benchmark approach, the basic architecture is the same.

1.4.4 Limitations of these techniques

Application, benchmark, and packet train analysis all rely on sending data across the network to obtain the measurements needed for performance prediction. This chapter has already described many of the factors in network design and use that come into play in determining network performance. Gauging network performance by sending data has the advantage of treating the network like a black box, which avoids these complexities. But there are a number of limitations that are shared by these three techniques.

Scalability

Sending data between two machines is an excellent way of measuring the network's performance between those two machines. Unfortunately, applications that can choose between many machines require more information. Examples include:

- selecting one of several machines for a long-running application,
- selecting the best n machines for a parallel computation,
- a real-time scheduling application that forwards tasks to the best-available machine, and
- selecting machines in a collaborative environment where information is needed for choosing servers to work together with several desktops distributed across the network.

Each of these situations requires knowledge of the network performance between more than a single pair of machines. The minimal amount of information required to solve these problems is the network performance between each pair of potential servers, and between each desktop and all of the servers. Because modern operating systems allow desktop machines to function as servers, these problems can only be solved with knowledge of the communication performance between all pairs of machines. If P is the number of machines in the system, this is an $O(P^2)$ problem. Even for the number of machines that might be found in a small department, perhaps $P = 100$, taking a measurement of network performance every five minutes would require over thirty measurements per second. Even if that is a tolerable fraction of the network's bandwidth, it makes it impossible to take a snapshot of the entire network simultaneously, or to observe the dynamic behavior of the LAN over shorter time intervals.

By grouping machines by location, it is possible to perform the benchmarks more efficiently [48]. For example, in the network shown in Figure 1.4, the network has been divided into the WAN, enclosed by the circle, and three LANs outside the circle. Measurements of the WAN's performance can be taken between the three machines on the circle, or between other machines in the different LANs in the common case where there is no congestion on the LAN worse than that in the WAN. In each individual LAN, measurements of the performance between the machines within that LAN, as well as the border machine, can be taken. These measurements can then be aggregated to provide a complete picture of network performance between machines in all three LANs.

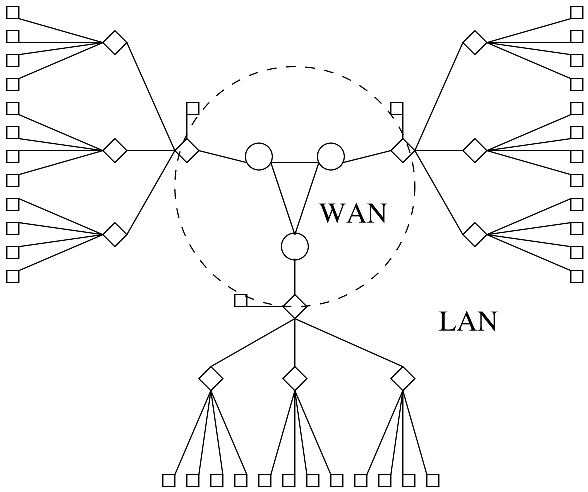


Figure 1.4: Several LANs joined by a WAN. The hierarchical structure of networks lends itself to partitioning performance measurement in a hierarchical fashion.

These techniques may be effective in some cases, but not in all. First, machines must be placed appropriately to measure across the WAN. Although unusual, if there is a bottleneck between the selected machine and the WAN, the measurement may not be useful to the other machines in the network. Secondly, even this solution does not make the technique scalable. The technique may allow larger number of machines at each site, but the overall technique is still $O(P^2)$ in the number of sites across the WAN. Placing machines at appropriate locations in the network to scale this solution across large numbers of machines in terms of both the number of machines at each site and the number of sites is very difficult.

Invasiveness

Another fundamental problem of measuring network performance by sending data across the network is that it quite naturally disturbs the system its measuring. Obtaining a prediction that 10Mbps is available on the network is useless if the measurement system is using that 10Mbps 25% of the time. Similarly to the uncertainty principle in quantum mechanics, a field widely believed to be as nebulous as networking, it is impossible to measure a network's performance without disturbing it in some way.

The packet train approach is an attempt to measure a network's performance with less disturbance to the network's capacity. However, it is unclear that the large number of samples needed to obtain an accurate prediction make it ultimately less invasive.

Topology

Finally, none of these measurement techniques provide information about topology. At first glance, knowing a network's topology may seem useful only as a way of improving the

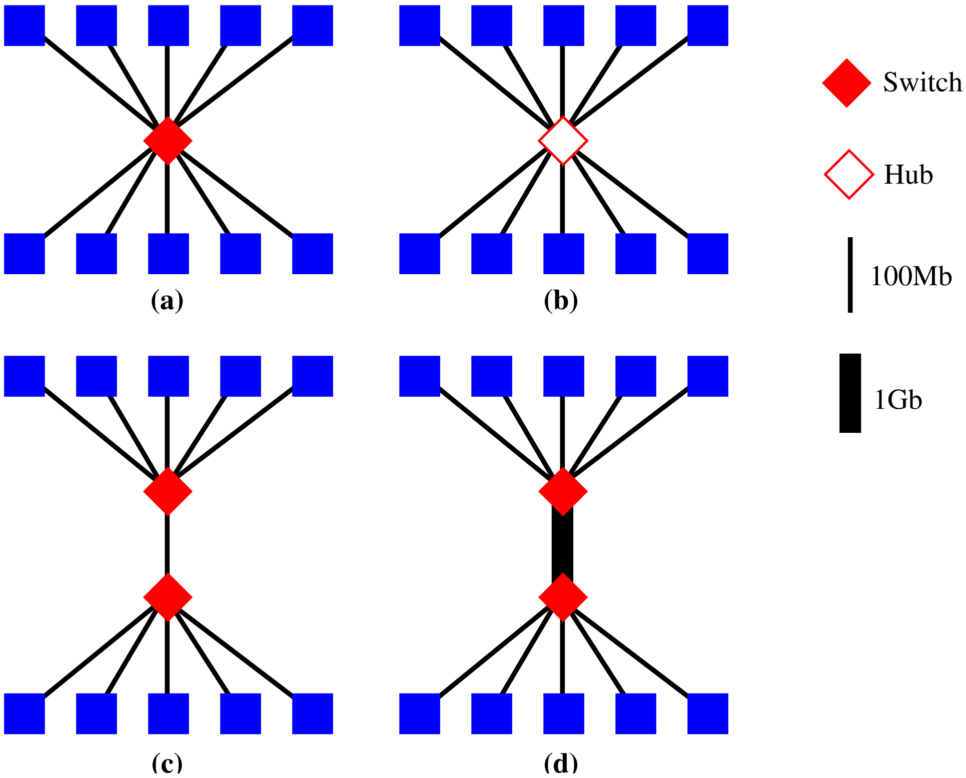


Figure 1.5: Four networks used to connect ten machine, each of which produces 10Mb for a single measurement, but whose performance varies widely when used in parallel.

scalability of the predictions. However, it is much more important for accurate predictions of the performance of parallel applications.

Consider the networks shown in Figure 1.5. These networks demonstrate four different ways in which it is possible to connect ten machines using simple off-the-shelf hardware. In each case, the bandwidth measured between any single pair of machines will be 100Mb. However, if these machines are used by a parallel application sending data between multiple pairs of machines simultaneously, their performance will vary tremendously. In Figure 1.5(a), a switch is used to connect the machines, so the full 100Mb bandwidth is available to each machine. Figure 1.5(b), however, represents a network where a hub was used instead of a switch. This architecture saves money, but now all machines share the same logical segment of Ethernet, so they share the same 100Mb bandwidth. In Figure 1.5(c), two different switches have been used to connect the two halves of the network. Here, the full bandwidth of the network can be used if each set of five machines is communicating internally, but if multiple messages are communicated between the two network halves, performance will be reduced. This bottleneck is removed in Figure 1.5(d), where a gigabit Ethernet link is used to connect the two switches, instead, again eliminating the bottleneck.

Figure 1.5 is an excellent example of situations where benchmarking cannot provide the information needed to support parallel applications. Technically, these problems can be overcome by performing multiple simultaneous benchmarks to determine whether there are correlations in the performance of separate connections. Unfortunately, this also raises the complexity of the technique to an infeasible $O(P!)$.

Manually providing the topology to the benchmarking tool allows for application-level sharing to be predicted and could solve the problems with some networks. However, even with knowledge of the topology, sometimes the performance of an application cannot be predicted with simple end-to-end benchmarks. In Figure 1.5(d), for instance, if applications running on other nodes unseen in the network are consuming 700Mb of the gigabit link, there is no way that an end-to-end benchmark between a single pair of machines can deduce that there will now be a bottleneck there if all ten machines are communicating in parallel.

This behavior is a downside to the beauty of networking protocols. Modern network architecture is completely transparent to the end-user. The network-as-a-black-box design has allowed great advances in simplifying the development of distributed applications, but makes it virtually impossible for the end-user to optimize network performance. Similar lessons have been learned from other systems that provide simple interfaces to complex systems. Consider advanced SMP systems, such as the SGI Origin. The complexities of managing the distributed memory are completely hidden from the user, making it possible for anyone to write a high-performance shared-memory program. However, to achieve optimum, or in some cases acceptable, performance, it is still necessary to write the program taking the architecture of the system into account.

1.5 Network-based measurement

To maximize the usefulness and portability of distributed applications, these three limitations must be overcome. Fortunately, there is an alternative. Rather than using end-to-end measurements to determine the performance of the network, consider using the status of the network components themselves to predict the performance of the end-to-end application. Although this is a more complex solution requiring several steps to reach the desired prediction of application performance, there are advantages that make up for the complexity.

Scalability Rather than having complexity of $O(P^2)$, this technique has only linear complexity in the number of network components, or $O(|\mathcal{N}|)$. Given the hierarchical structures used by networks, this complexity expands to $O(P \log P)$, which is manageable over extremely large networks.

Invasiveness If the amount of traffic passing through the network can be obtained from the network devices themselves, then their internal monitoring capabilities can be used to do the actual work, and the only additional traffic traversing the network will be the queries. Assuming these queries are reasonably efficient, this technique should not disturb the network significantly.

Topology Obviously obtaining the network's status directly from the network results in detailed knowledge of the network's topology, needed for predicting the performance of parallel applications that send multiple messages simultaneously.

A snapshot of the network $\mathcal{N}$, consisting of the status of all parts of the network at the same instant, is denoted N. Using a history-based prediction of the network snapshot, N_t, an application's performance can be predicted as $A_n(N_t)$.

The three techniques described here for obtaining a prediction of an application's future performance running on $\mathcal{N}$, denoted $\hat{A}(\mathcal{N})$, are described by the following equations:

$$\hat{A}(\mathcal{N}) \approx \begin{cases} A_t(\mathcal{N}) & \text{Application-based} \\ A_n(N_t) & \text{Network-based} \\ A_n(M(B_t(\mathcal{N}))) & \text{Benchmark-based} \end{cases}$$

$\mathcal{N}$	real network
N	network status snapshot
A	application
B	benchmark
$A(\mathcal{N})$	performance of A running on $\mathcal{N}$
$A_t()$	time series performance prediction of A
$A_n()$	network model performance prediction of A
$M()$	mapping function inferring network status from benchmark performance

Figure 1.6 conceptually illustrates how these techniques interact with the network to predict application performance.

1.6 Using network-based prediction

The shortcomings of the application-level approaches to network performance measurement and application optimization motivate a new approach to the problem of network analysis and prediction. Rather than measuring the network's behavior at the application level, I propose to characterize the network at the lower component level. I will demonstrate that *the low-level details obtained from network components provide the information needed by distributed applications to adapt themselves to modern network environments.*

Low-level information solves the problems inherent with the application-level approaches. The low-level information

- provides a scalable solution to the performance prediction problem,
- can be obtained with minimal overhead, and
- offers the topology information needed for adaptation.

To demonstrate that providing low-level network information to applications is both feasible and useful, I address a variety of issues in this dissertation.

Application requirements A wide variety of applications benefit from the knowledge obtained through low-level information about the network's structure and status. The

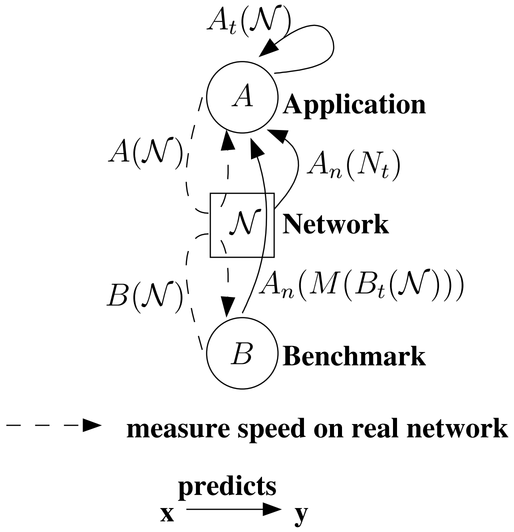


Figure 1.6: Conceptual diagram of options for prediction. The dashed arrows illustrate the actual application being run on the network as a probe of its status. The solid arrows illustrate the conceptual paths taken to predict an application's performance using data obtained by running the application itself, by obtaining information directly from the network, and by using a benchmarking program to determine the network's performance.

importance of topology information to collective communication was described in Section 1.3. Processor selection, discussed in Section 4.3, is another common need of distributed applications that can be done well using low-level information. The particular knowledge available at the network level also allows better models to be built for application performance, such as that discussed in Section 4.4. By showing that common operations benefit from low-level information, I demonstrate the true general application that low-level network information has to many applications.

Interface The low-level information can be made available through a portable, high-level interface. Chapter 2 describes Remos, a system I have co-designed that provides the needed information to applications. The most significant challenge in the design of Remos is providing information about low-level network behavior without sacrificing the portability enjoyed by distributed applications. Remos maintains portability by providing a best-effort virtual representation of the network's topology, attempting to represent the behavior of the network over its actual structure. To support applications with simpler needs, Remos also provides flow queries that allow the application to obtain predictions of application-level performance without analysis of the network topology or a loss of information in the conversion to the portable representation.

Shortcomings of high-level techniques My thesis, that low-level knowledge is the best way to predict and optimize application-level performance, runs counter to a great amount of common practice and belief. Networking is generally treated like a black box at the application layer. Although networking research is becoming more common in recent times, few people have proposed breaking this boundary down, preferring to offer advanced network services, but maintain the fundamental boundary. I have already discussed some of the techniques used for network measurement. After describing the network-based techniques in more detail, I will discuss some of the differences, particularly in terms of scalability and invasiveness, in Section 3.6.

Providing end-to-end predictions Although the topology and scalability advantages of using low-level information are clear, the more challenging questions arise when considering how to provide predictions of application-level performance relying on the low-level knowledge. Chapter 3 describes the network-based performance prediction technique and presents several experiments demonstrating that its accuracy is similar to that of application-based approaches.

Topology discovery The topology information available through the low-level network information is one of the most important advantages of this approach. Chapter 4 describes how topology information can be acquired using SNMP. At the IP level, topology information is quite easy to extract, but level 2 networks are frequently more challenging. Much of this chapter is dedicated to a description of how to determine the topology of the bridged Ethernet, the most common local area network in use for the past decade. One of the most significant challenges in designing this algorithm is dealing with the incomplete forwarding knowledge that is available from the bridges. I present an algorithm which is provably good when faced with incomplete information.

Feasibility The low-level approach to network characterization and performance measurement is implementable today. Both Chapters 3 and 4 describe techniques that have been implemented and tested using modern commodity hardware. Although access to low-level knowledge of wide area networks is typically limited, due to business concerns, there is no technical reason why this approach cannot be implemented today.

The core question my dissertation addresses is whether the violation of the end-to-end systems principle is warranted for network characterization and performance prediction. I will show that it is both necessary to violate this principle and possible to do so with minimal sacrifices. To answer this question, I take a top-down approach, beginning with why access to low-level information is needed at the application layer and concluding with how it can be accomplished at the network layer.

Chapter 2

Remos

Remos was developed to meet the resource information needs of adaptive applications. It sits between the application and the variety of networking resources, computing resources, and software systems used to control them. The principle design goal of Remos was aggregation of the variety of information available from these systems into information accessible and useful to as many applications as possible. This goal necessitated the development of a standardized way of representing data about the capabilities of the networks supported by Remos.

This chapter addresses the question of how low-level network information can be made available to high-level applications in an accurate and portable manner. Such an interface must support both low-level information, such as topology and link-by-link capacity and utilization, as well as high-level information, such as end-to-end flow bandwidth and prediction of future performance. Remos addresses these issues, although compromises are necessary in places where maximizing portability is at odds with maximizing the accuracy of the low-level information. The diverse interests of the people involved in the development of Remos has helped ensure that its design meets the needs of a wide variety of applications.

The Remos system is the product of work by a large number of people. As one of the original designers of the interface, and the most outspoken proponent of the topology interface, my contributions to it have included the initial proposal for the topology interface, working with a small group to combine the various ideas into a coherent interface, and participating in many design meetings with other members of the Remulac project. I also designed the original collector-modeler protocol and designed and implemented the bridge collector. Much of the content of this chapter is based on my own ideas, but there is no part of Remos that is not made up of the contributions of many people.

2.1 Overview

Networked systems provide an attractive platform for a wide range of applications, yet effective use of the resources is still a major challenge. A networked system consists of compute nodes (hosts), network nodes (routers and switches), and communication links. Network conditions change continuously due to the sharing of resources, and when re-

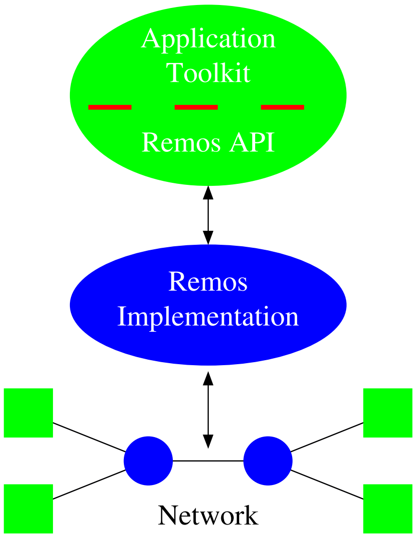


Figure 2.1: Remos is divided into two primary components. The Remos API, which is described in this chapter, is provided as a library applications can link with to obtain the network information they require. The actual data gathering component of the Remos architecture exists as a suite of separate programs and will be described in Section 2.9.

source demands exceed resource availability, application performance suffers. Congestion on network nodes and links can reduce the effective bandwidth and increase the response time observed by applications. On compute nodes, competing jobs and higher-priority activities reduce availability. An attractive way of dealing with such changes is to make applications *system-aware*, i.e., the application periodically adapts to the system in an application-specific manner.

The Remos system provides applications with a query-based interface to their execution environment including the network state. It is designed for experiments with the coupling of network-aware applications and network architectures. The Remos system has its roots in two separate projects: an investigation of resource management in application-aware networks (Darwin) and an effort to support system-aware applications through libraries, frameworks, and tools (Remulac).

Network-aware applications must be able to obtain information about resource availability, in particular, the network's capabilities and status. Unfortunately, network architectures differ significantly in their ability to provide such information to a host or to an executing application. To avoid dependences on the idiosyncrasies of network architectures and communication systems, application development for networks requires a system-independent interface between applications and networks. A uniform interface allows the development of portable applications that can adapt on a range of network architectures. Furthermore, a system-independent interface is crucial for allowing applications to adapt on heterogeneous networks, where components are realized with different network technologies.

The general architecture of Remos is depicted in Figure 2.1. The Remos API, which is the principle topic of this chapter, exists in the middle of this architecture. Above it lie the many different applications that may make use of the information made available through the Remos API. Applications may communicate directly with the Remos API, or they may be developed using libraries or other middleware that uses Remos internally to provide services to the application. Below the Remos API lies the implementation of Remos needed to gather the information to answer queries made to the API. The architecture of this will be briefly described in this chapter, but details about the measurement techniques used to provide this information will be left for later chapters.

As indicated by Figure 2.1, Remos lies between the network-level hardware and the application layer. The purpose of this chapter is to demonstrate that it is possible to take low-level information and provide it in a portable format suitable for use by applications. Making the transition from low-level to high-level is crucial for taking advantage of the information available at the network-level.

2.2 Design challenges

In this section we discuss the problems that a common, portable interface like Remos must address. In the next section we present the design for the Remos system and explain how it addresses the challenges presented here.

2.2.1 Dynamic behavior

We must characterize network properties that can change very quickly. Moreover, application traffic can have widely different characteristics, so applications may want access to different types of information. For many data intensive applications, the burst bandwidth available on a network may be more important than the average bandwidth. In contrast, applications that stream data on a continuous basis, such as video and audio applications, may be more interested in the available bandwidth averaged over a longer time interval. The Remos interface should satisfy these diverse requirements.

An application is most interested in the expected traffic on the network in the future. A preview of future properties would allow the application to adjust to the actual situation encountered in the next t units of time. Unfortunately, information on future availability of resources is impossible to find in general, although some sophisticated network management systems may be able to provide a good estimate based on the current knowledge of applications and their resource usage. This is an ongoing research problem.

2.2.2 Sharing

Connections (as seen by the applications) will usually share physical links with other connections of the same and other applications. This dynamic sharing of resources is the major reason for the variable network performance experienced by applications, and Remos should consider the sharing policy when estimating bandwidth availability.

An important class of applications that Remos attempts to support is parallel and distributed computations that simultaneously transfer data across multiple point-to-point connections. This kind of traffic pattern is typical of large-scale scientific computations (when mapped onto a distributed system) and distributed simulations. Since multiple parallel data transfers may be competing for the same resources, Remos should consider all exchanges in a data transfer step collectively rather than separately for each pair of endpoints. Again, this will require characterizing sharing behavior.

Determining a solution to the problem of characterizing the performance of multiple simultaneous data transfers is complex due to the interactions between the specific topology, network sharing policies, and the timing of messages.

2.2.3 Information diversity

Most installed networks do not have facilities that can directly provide the information applications need to adjust their resource demands. For that reason, highly portable, standard protocols such as TCP/IP rely on indirect mechanisms to obtain information about the network status, for example, dropped packets indicate congestion.

The information that may be of interest to applications includes static topology, routing, dynamic bandwidth (possibly averaged over different time intervals), and packet latency. Different types of information are generated by different entities, are maintained in different formats and locations, and change on different time scales. For example, some information may only be available through a static database, perhaps maintained off-line by a system administrator. Other information may be accessible in a systematic fashion using protocols such as SNMP [25]. Finally, some information, such as packet latency, is not routinely collected and may have to be measured directly by the Remos system using benchmarks specifically developed for this purpose.

2.2.4 Heterogeneity

Networks differ significantly in how much information they collect and make available. For example, dynamic link utilization of point-to-point links connecting routers can often be obtained through SNMP. However, shared Ethernets typically do not have a single entity collecting information on network utilization. Some networks do provide very specific feedback to endpoints on available network bandwidth as part of traffic management (flow control). The most important example is Available Bit Rate (ABR) traffic over ATM networks, where rate-based [18] or credit-based [26, 70] flow control tells each source how fast it can send. This information is currently only used at the ATM layer, but could be made available to higher-layer protocols or applications.

The challenge in designing a portable interface is to find a way to cover the entire range from currently deployed networks such as shared Ethernets with very large numbers of users, to more advanced commercial networks such as ATM. For an interface to be useful, it is not necessary that all information is available for every network. Sometimes partial information, such as static link capacities, may have significant value to applications.

However, dealing with partial information is likely to make application development more complicated.

2.2.5 Level of abstraction

One of the thorny issues in designing an interface between applications and networks is deciding what aspects of the network should be exposed to applications. This problem shows up in a number of contexts.

One issue is heterogeneity. As discussed earlier, hiding network-specific details is important for portability, but it is not always clear how to do that without losing important information. A second example concerns limiting the volume of information provided to the application. Since all relevant hosts may be connected to the world-wide Internet, we need a way to limit the amount of information returned to an application, since providing information on the entire Internet is both unrealistic and undesirable. In some cases the scope of the query can be limited easily. For example, applications restricted to a LAN will only need information about the LAN. In other cases, solutions are less obvious. For example, how do we limit information for an application using three hosts, one located at Carnegie Mellon University, a second at ETH in Zurich, and a third on a plane flying from the US to Japan?

Another important problem is management of routing information: if there are multiple paths between two hosts A and B, a network architecture may use different paths for individual data units (packets) traveling from A to B, and these paths may exhibit vastly different performance characteristics. Exposing this detail to the application is problematic since it is at a low level and changes rapidly. Furthermore, current protocols do not allow applications to influence routing, or even request that routing remains fixed. Therefore the fact that there are multiple physical paths, and that specific routing algorithms are responsible for performance differences and changes, is of limited value to applications. On the other hand, hiding the information is not without problems if the two paths differ in bandwidth or congestion.

All these problems center around the same question: what is the right level of abstraction for network information provided to the application. One option is to present the entire network topology, along with routing information and the characteristics of each link and node on the network. Such a description would provide all possible information, but it includes many details that are not relevant to most users of Remos, and it may be difficult to interpret. The other extreme is to provide the information at a much higher level, focusing on the performance characteristics that may be of interest to the application. This interface would be easier to use and the problem of information overload is avoided. However, in some situations, this may lead to information being vague and inaccurate, and potentially useless to applications.

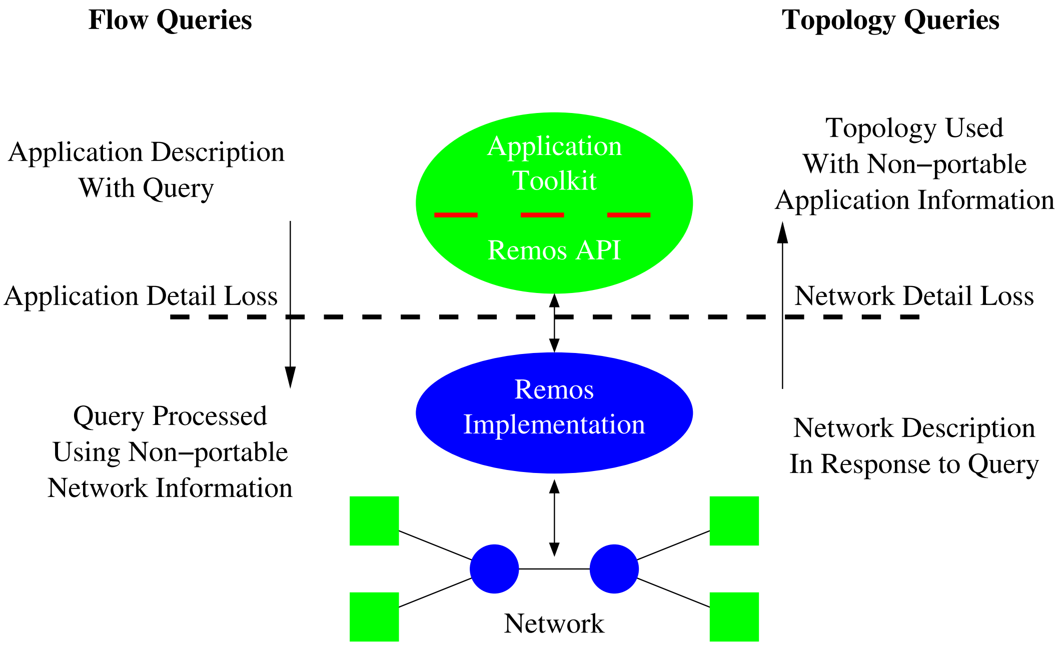


Figure 2.2: The two query abstractions supported by Remos are illustrated here. As each query passes through the abstraction layer between the application and network levels, information is lost. A user should select the best query for an application by evaluating the complexities of the application's adaptation options and whether any unusual network support might be available that is not reflected in the standardized topology description.

2.3 Remos design principles

The guiding principle in the development of Remos was to provide as much information about the network as possible without sacrificing portability of the interface. There is an obvious conflict, however, between maintaining portability and maximizing the information presented.

The basis of this conflict is inherent in the standardization process—both applications and networks are sufficiently diverse that there is no clear way to describe all of their capabilities. There is information loss whether the network description is standardized for use at the application layer, or if the application description is standardized for use at the network layer. There is even more loss if both descriptions are standardized for a totally independent scheduler. Because we wished to minimize information loss while maximizing portability, we elected to support two different types of queries in Remos. The two query types are illustrated in Figure 2.2.

The first, flow-based queries, is used when the application itself is fairly simple, or wants to evaluate the performance a particular communication pattern will receive from the network. These queries require a standardized description of a communication pattern to be used by the application. This introduces information loss, but if the application's communication needs are simple, that loss should be minimal. The standard description is then passed to the network layer, which is free to use whatever network-specific knowledge it has to respond to the query.

The second type of query is the topology query. The topology query is useful for the opposite problem, when an application is rather complex, and its options for network utilization are too complex or would take too many separate queries to evaluate using the flow-based queries. In the topology query, the network's representation, including topology, link capacity, and utilization, is passed to the application layer in a standardized format. Again, this process introduces information loss, but it enables applications to make decisions such as task placement without incurring exponential costs.

The following sections will describe the two query styles in detail and review their advantages and disadvantages.

2.4 Flow-based queries

A flow is an application-level connection between a pair of computation nodes. Using flows instead of physical links provides a level of abstraction that makes the interface independent of system details. All information is presented in a network-independent manner. While this provides a challenge for translating network-specific information to a general form, it allows the application writer to write adaptive network applications that are independent of heterogeneity inherent in a network computing environment. We will discuss several of the important features of the flow-based query interface in the remainder of this section.

2.4.1 Multiple flow types

Applications can generate flows that cover a broad spectrum. Flow requirements can range from fixed and inherently low bandwidth needs (e.g. audio), to bursty higher bandwidth flows that are still constrained (e.g. video), to unconstrained flows that can consume any available bandwidth, but may not have any quality requirements. Different flow types may require different types of queries. For example, for a fixed flow, an application may be primarily interested in whether the network can support it, while for an unrestricted flow, the application may want to know what average throughput it can expect in the near future.

Remos collapses this broad spectrum to three types of flows. A first type consists of *fixed flows* that require a specific bandwidth. A second type consists of *variable flows*. Flows in this category can use larger amounts of bandwidth, and the bandwidths of the flows are linked in the sense that they will share available bandwidth proportionally. For example, three flows may have bandwidth requirements of 3, 4.5, and 9 Mbps relative to each other; the result of a corresponding Remos query may be that the flows will get 1, 1.5 and 3 Mbps respectively. A third type consists of *independent flows*, for which the user would like to know how much bandwidth is available after the requirements of the first two classes have been satisfied. Each independent flow is considered independently. These can be viewed as lower priority flows, or used to select between several choices the application has for communication.

Simultaneous flow queries may specify flows of any types. The combination of different flow types allows a wide variety of situations to be described concisely.

2.4.2 Simultaneous queries and sharing

Flows may share a physical link in the network. At bottleneck links, this means that flows are competing for the same resource, and each flow is likely to get only a fraction of the bandwidth that it requested. Of particular interest is the case where multiple flows belonging to the same application share a bottleneck link. Clearly, information on how much bandwidth is available for each flow in isolation is going to be overly optimistic for all flows together. Remos resolves this problem by supporting queries for both individual flows, and simultaneously for a set of flows. The latter allows Remos to take resource sharing across application flows into account. Support for simultaneous flow queries is particularly important for parallel applications that use collective communication.

Determining how the throughput of a flow is affected by other messages being sent at the same time is very complicated and network specific. A variety of different sharing algorithms that affect the proportion of bandwidth received by a particular flow are deployed. While some networks have sharing policies that are precisely defined for certain types of traffic (e.g. ABR flows over ATM, or flows with bandwidth guarantees over FDDI II or ATM), on other networks (e.g. Ethernet), full characterization of sharing behavior would require consideration of packet sizes, precise packet timings, queueing algorithms, and other factors. Moreover, how much bandwidth a flow gets depends on the behavior of the source, as senders can vary in their aggressiveness and how quickly they back off in the presence of congestion.

Because of the complexities involved, the Remos implementation has not attempted to characterize these interactions accurately in general. Our approach is to return the best knowledge available to the implementation that can be returned in a network-independent manner. In general Remos will assume that, all else being equal, the bottleneck link bandwidth will be shared equally by all flows (not being bottlenecked elsewhere). If other better information is available, Remos can use different sharing policies when estimating flow bandwidths. The basic sharing policy assumed by Remos corresponds to the max-min fair share policy [59], which is the basis of ATM flow control for ABR traffic [4, 60], and is also used in other environments [53].

My research has examined options for overcoming these restrictions in future revisions to the Remos implementation, or other network measurement systems. I have found that, although the problem is still imposing, useful information can be determined using relatively simple models. To date, these models have not been implemented in Remos itself, therefore discussion of these models will be postponed to Chapter 3.

2.4.3 Example

To see how simultaneous flow queries can be made using multiple flow types, consider the illustration in Figure 2.3. The middle diagram shows how the bandwidth of the fixed flow is first removed from the network. Next, the two variable flows share the bandwidth avail-

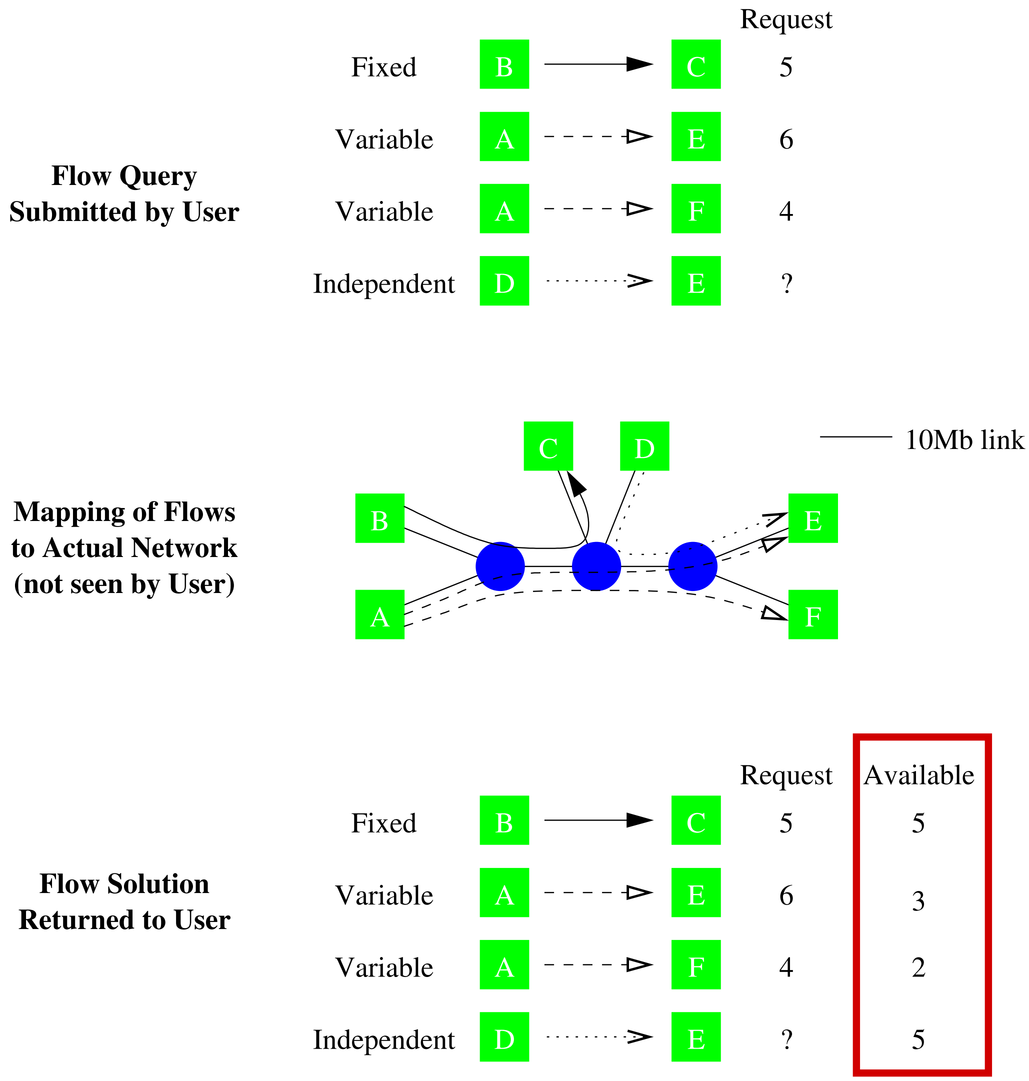


Figure 2.3: A flow query submitted to Remos, shown with the corresponding mapping of the flows to the physical network and the solution returned to the user.

able at their shared bottleneck. Finally, the independent flow is allocated the bandwidth remaining along its path. In this case, that turns out to be more than the variable flows.

2.5 Topology queries

Remos supports queries about the network structure and topology in addition to queries about specific flows in the network. The reason we expose a network level view of connectivity is that certain types of questions are more easily or more efficiently answered based on topology information. For example, finding the pair of nodes with the highest bandwidth connectivity would be expensive if only flow-based queries were allowed.

Graphs are a well-accepted representation for network topology. Remos represents the network as a graph with each edge corresponding to a link between nodes; nodes can be either compute nodes or network nodes. Applications run only on compute nodes, and only compute nodes can send or receive messages. Network nodes are responsible only for forwarding messages along their path from source to destination. Each of the communication links is annotated with physical characteristics such as bandwidth and latency.

Topology queries return the graph of compute and switch nodes in the network, as well as the *logical* interconnection topology. Use of a *logical topology graph* means that the graph presented to the user is intended only to represent how the network behaves as seen by the user—the graph does not necessarily show the network's true physical topology. The motivation for using a logical topology is information hiding; it gives Remos the option of hiding network features that do not affect the application. For example, if the routing rules imply that a physical link will not be used, or can be used only up to a fraction of its capacity, then that information is reflected in the graph. Similarly, if two sets of hosts are connected by a complex network (e.g. the Internet), Remos can represent this network by a single link with appropriate characteristics.

In the absence of specific knowledge of sharing policies, we recommend that users of logical topology information assume that bandwidth is shared equally between flows. This assumption can be verified using queries. If other sharing policies become common, we could add a query type to Remos that would allow applications to identify the sharing policy for different physical links.

Many networks, such as Ethernet, offer both full- and half-duplex links with which to build the topology. In current Ethernet networks, for instance, frequently the connection between bridge and CPU is half-duplex, while the links between switches are full-duplex. Furthermore, routing is not necessarily symmetric, so paths connecting two sets of machines may follow a completely different set of switches in each direction. The result is that three types of links: full-duplex bidirectional, half-duplex bidirectional, and unidirectional are commonly used in networks.

Because these characteristics make a dramatic difference in the performance of a network, Remos needs to support accurate representations of the duplex of each network link. Furthermore, because the graphs will typically be analyzed by other algorithms, the representation must be easy to process automatically. We rejected merely adding flags to each link to indicate whether it is full- or half-duplex, primarily because doing so would intro-

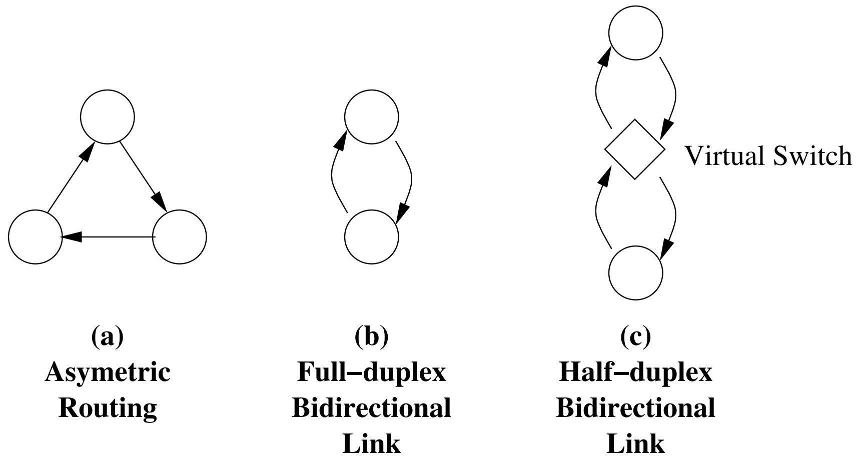


Figure 2.4: Representation of asymmetric routing, full-duplex, and half-duplex bidirectional links with unidirectional links. In (a) and (b), the links themselves are used to indicate the bandwidth characteristics in each direction. In (c), because the same link capacity is shared in each direction, a virtual switch is inserted instead. The internal bandwidth and latency of the virtual switch is used to represent the capacity of the half-duplex physical link, while the four unidirectional links in the logical topology are set to a higher bandwidth and zero latency.

duce challenging programming issues to select the correct capacity and utilization when the link may be traversed from both ends.

The solution adopted by Remos is to use unidirectional links exclusively. Both full-duplex and half-duplex bidirectional links can be represented using combinations of these unidirectional links and virtual switches. Again, the logical topology given to the application may not represent what is physically present in the network, but the functionality remains the same. And, in this case, there is no way for the application to know what hardware is being used to build the network. Figure 2.4 depicts how unidirectional links are used to address the issues of asymmetric routing, full-duplex bidirectional links, and half-duplex bidirectional links.

Although Remos uses several constructs to represent different types of physical links, adding them to topology drawings introduces a great deal of clutter, without conveying any additional useful information. In this dissertation, therefore, network drawings will continue to use single links between nodes, with the implicit assumption that each link is full-duplex bidirectional, unless otherwise mentioned.

All information is represented in a network and system independent form. Hence, the network topology structure is completely independent of peculiarities of various types of networks and manages network heterogeneity in a natural way. Remos emphasizes information that can be collected on most networks, but it should be noted that not all types of information may be provided by all networks and Remos implementations. For example,

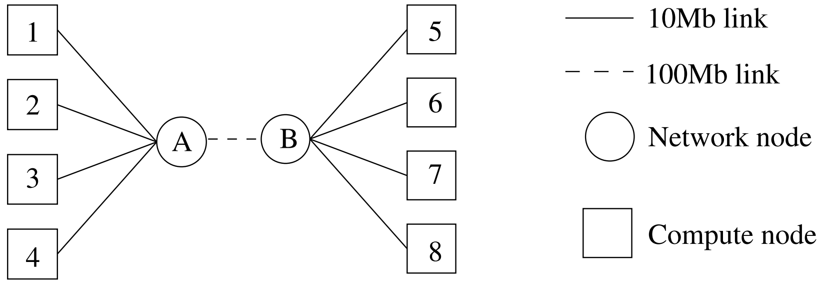


Figure 2.5: Remos graph representing the structure of a simple network. Nodes A and B are network nodes, and nodes 1–8 are compute nodes.

if a network architecture provides specialized information like reliability or security of a link, Remos could be extended to include such information when available.

2.5.1 Examples

Figure 2.5 shows a simple network represented by a graph. The links in this network representation are annotated with network performance information. However, it is just as important that the nodes include performance information as well. For instance, if nodes A and B each have an internal bandwidth of 80Mbps and all the compute nodes have bandwidths higher than 10Mbps, then the links connecting the compute nodes to the network nodes restrict bandwidth, and all nodes can send and receive messages at up to 10Mbps simultaneously. On the other hand, if nodes A and B have internal bandwidths of 10Mbps, then these two network nodes are the bottleneck and the *aggregate* bandwidth of nodes 1-4 and 5-8 will be limited to 10Mbps.

In the previous paragraph we (implicitly) assumed that Figure 2.5 represents a physical topology consisting of 8 workstations, 2 routers, and 9 links. However, Figure 2.5 can also be interpreted as a *logical* topology, potentially representing a broad set of (physical) networks. For example, if A and B have internal bandwidths of 10Mbps, it also represents two 10Mbps shared Ethernets, containing nodes 1–4 and 5–8 respectively, that are connected to each other with a 100Mbps link. While it may not correspond to the physical structure of the Ethernet wiring, it accurately represents its performance.

The importance of the distinction between compute and network nodes is illustrated in Figure 2.6. In Figure 2.6(a), a high speed “research” network has been added to compute nodes 1–4. Similar to the way most research networks are deployed, the nodes are dual homed (i.e., they are simultaneously connected to both the research and the “regular” network). Because compute nodes cannot forward messages, node 1, for instance, cannot use the research network to reach any node other than nodes 2–4. If connectivity to other nodes is desired, one of the compute nodes can also serve as a router, as is logically shown in Figure 2.6(b). With this representation, node 1 can receive 10Mbps of data from node 5, and at the same time, send 10Mbps of data to node 6 through nodes C, D, A and B.

The sample network of Figure 2.6 also brings up the complications of routing. With simple networks such as the one shown in Figure 2.5, only one path connects every pair of

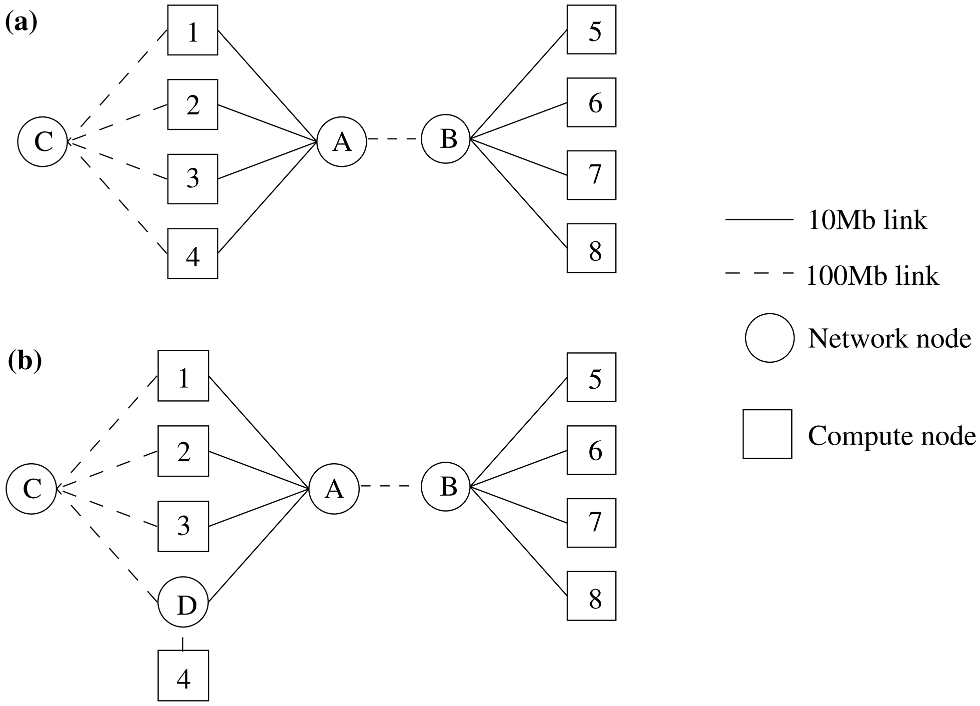


Figure 2.6: Network from Figure 2.5 with additional network added to nodes 1–4. (a) no routing between the two networks; (b) Node 4 has been modified to also route messages between the networks.

nodes. However, in Figure 2.6(b), there are two possible routes between nodes 1–3 and the rest of the network. Because of the complexity of the routing issues, Remos does not export the routing information to applications, but may use this information to answer flow-based queries. Many networks have an acyclic topology, as shown in Figure 2.5. If multiple paths are possible, then only one path is likely to be of interest at a time, and other paths will not even show up in the logical topology. For example, most applications would only use the research network, or the regular network, but not both at the same time, so the Remos interface would only return one of the two subnets, depending on the scenario.

Capturing routing information in the model would add a lot of complexity, while rarely adding value, so we decided not to include it. Either a large set of parameters or an entire routing description language might be required. A simple table can handle static routes, but many networks where routing decisions are necessary make dynamic decisions based on network traffic, packet size, volume of data from a single host, host priorities, application priorities, or at the control of the application. Until a better description of how these issues can be handled in a portable manner is available, the topology interface simply avoids including this information.

Remos may return a topology graph with multiple routes, when they are feasible, but no other routing information is returned. We expect that an application that makes use of such topology graphs is informed about how to properly interpret the information. Note that the results of both flow-based and topology queries implicitly include a lot of routing information.

Although the routing information is not available through the topology graphs, the query interface for simultaneous flow operations can provide accurate feedback on what happens, given a particular communication pattern. The Remos Modeler can query the Collectors for the route used by each separate flow, which allows the correct routing information to be used when the routing is statically determined.

2.6 Limitations

Our design of Remos ignores a number of important network properties. As we just mentioned, it deals with the issue of alternate routes in a restricted way. Moreover, it does not deal with multicasting, or with networks that provide guaranteed services. While both of these are important features that would be of interest to applications using Remos, the networks deployed today rarely make this functionality available to applications. As application-level multicast and guaranteed services become more widely available, we will extend Remos to support them.

2.7 Application programming interface

The Remos API is divided into three classes of functions: status functions, fitting functions, and topology queries. We briefly outline the main functions in this section.

All queries support a timeframe field to specify the timeframe of interest to the application. Applications can specify queries that return information on previous, current, or a prediction of future network conditions. Physical characteristics can also be requested.

2.7.1 Status functions

Status functions return information on individual compute nodes and simple node-node flows. An additional function in this category allows the user to supply information on the software overhead contributed by the communication software to the Remos interface, so that such information can be taken into account when computing the responses to queries:

- **remos_get_node()** This provides information about a node's network characteristics.
- **remos_get_flow()** This provides information about the characteristics along a path between two nodes.
- **remos_node_delay()** This allows the user to contribute information about delays contributed by software layers unknown to Remos.
- **remos_get_node_info()** This call returns information about a node's compute power and load.

- **remos_node_query()** This call allows expandable queries to be made about a host. For example, currently a query can be made to obtain a list of shared libraries available on a machine, to ensure that an application can execute on it.

2.7.2 Fitting functions

Fitting functions return information about the ability of a network to support several simultaneous flows. They implement the concept of flow-based queries with sharing discussed in Section 2.4.2. Fitting functions allow an application to determine the service that will be received by a new set of flows, given the resource requirements of the existing flows and the sharing properties of the network (Section 2.2.2). The **remos_flow_info()** query allows the user to specify a set of end-to-end flows. The function then returns the bandwidth needs that can be met by the network. The **remos_flow_info()** query takes a set of parameters that can be used to describe a wide variety of scenarios.

- **fixed_flows** are flows that have specific bandwidth requirements that cannot be altered. If one cannot be met, the call will return failure, with an indication of what can be met. An important purpose of this flow class is that it allows the application to specify flows that the other two classes must compete against.

- **variable_flows** are used when an application is interested in determining how much bandwidth is available if it attempts to simultaneously send data in several flows. All flows in this class will be adjusted to rates that can be met by the network. The adjustment takes place in a approximately proportionate manner, based on the bandwidth requested for each flow and overall network constraints. A requested bandwidth is not reduced unless it is constrained by the network.

- **independent_flows** are used to determine how much bandwidth is remaining for additional flows, after meeting the first two needs. Each flow specified in the set is considered independently. An important use of this query is for deciding among several options for a planned communication.

This interface has the advantage that the user can obtain a proper answer for communication on networks with topologies and behaviors that may not be accurately representable through the topology interface.

A flag is provided to specify that a reservation should be placed for the resulting communication specification. The Remos interface does not support this reservation directly. However, when implemented on top of a network with QoS support, the reservation service would be a natural extension to Remos. A reservation request will be simply denied if such requests are not supported by the underlying network platform.

2.7.3 Topology functions

The third and final category is the topology query interface. This interface allows the user to obtain a network topology for a set of nodes selected by the user. As described in the design section, this topology is intended to represent the network's performance characteristics as seen by the application. Due to the variety of techniques for handling routing on networks, routing is not represented explicitly on the network. In most cases, a simple prediction based on the shortest path algorithm is appropriate, while in other cases, outside knowledge of the routing algorithm may be required.

2.8 Using the API

The usefulness of an API is determined by the types of applications that can make use of it. Several applications are presented throughout this dissertation. Here I present brief overviews of several other examples, as well as an example of how the Remos API is used in pseudocode.

2.8.1 Adaptive applications

Some real-time applications, such as distributed video systems, require consistent quality of service to perform effectively. However, many applications with weaker real-time properties can adapt to the available network performance, for example by varying their frame rate, image quality, and processing load [29].

One system that can make use of such information is Odyssey [86], which manages a variety of adaptive applications across networks ranging from high-performance to wireless systems. Odyssey combines protocol specific adaptations, such as frame rate and quality, with function and data shipping, to select the best combination of network and computational resources to achieve the desired application quality metrics. Adaptation in Odyssey is based on a Viceroy, a controlling authority for the resources on each node. Each Viceroy interacts with Wardens that implement the adaptation mechanisms for specific distributed resource types. The Viceroy is currently responsible for collecting information on network status, but instead, it could use Remos to retrieve this information. In fact, a similar modification has already been made to support the status of wireless networks [39].

Video applications are interested in flows between a client node and a small set of server nodes. The Remos fitting function is the most appropriate in this case since it allows the application to ignore the rest of the network and focus only on the aspects of the network that affect the client's performance. The Viceroy can issue a flow-based query for each of the servers or server combinations, and select the servers that give the best throughput. It can periodically reissue the query, or it can ask for a callback when conditions in the network change considerably.

Adaptation is also widely used in the area of high performance computing. The Fx compiler [109, 110] has successfully used automatic analysis to assign nodes to meet the performance requirements of applications containing multiple tasks. However, this analysis assumes static network behavior, and a system like Remos is required for adaptation in a dynamic network environment. Two other examples of network-aware applications include a pipelined application that adapts the pipeline depth [102], and simple distributed matrix multiply that selects the optimal number of nodes [111]. In both cases, applications used simple benchmarks to characterizing network performance. Remos provides a simpler and possibly more accurate alternative.

2.8.2 Clustering

A decision regarding what nodes to use for the execution of a parallel application can be made on the basis of availability, experience, performance, or efficient use of resources. Independent of the motivation, making these decisions requires some information on network and system performance. A typical problem is to locate the largest group of machines that have both a certain compute power and are capable of sustaining a certain bandwidth between themselves.

The network topology interface is the most appropriate mechanism to collect information to solve this problem. The interface can be used to connect candidate nodes with information on bandwidth availability on links connecting them. One of several known algorithms can then be used to deduce what nodes are "closest" together and meet connectivity requirements.

The code in this section demonstrates how the Remos API and data structures are used to build a cluster of tightly coupled machines using a greedy heuristic. This is not an optimal algorithm, but provides a good example of how the topology interface can be used to select nodes based on network performance.

The cluster is chosen as a set of n processors that are *close* to a designated node. The criterion for closeness is the average time to send a message of a given size to other nodes in the cluster. The input is a list of nodes, a designated start node, the size of the desired cluster, and a message size. The return value is a cluster of selected nodes. At each step, the routine looks at the nodes not in the cluster and adds the one that has the lowest average communication time with nodes already in the cluster. The communication times between nodes in the cluster and those not in it are calculated using `compute_distances`.

This pseudocode is written for intelligibility and is neither legal C nor Java.

```
/* make_cluster = routine to use greedy algorithm to form a cluster
 *    startHost: initial node to begin forming cluster around
 *      clusterN: number of nodes to be in cluster
 *         nodes: all candidates for the cluster (includes startHost)
 *        length: size of message for determining cost
 *
 *       returns: list of nodes in cluster
 */
Remos_Node_List make_cluster
(Remos_Node startHost,
         int clusterN, Remos_Node_List nodes, int length){
   RemosGraph graph;
   double timeframe = 0;
   Hashtable costFromNode[clusterN];
   NetworkCost zeroCost;

   Remos_Node_List clusterNodes; /*actual members*/
   closterNodes = remos_new_list(clusterN);
   clusterNodes.add(startHost);
```

```
   zeroCost = new NetworkCost();
   zeroCost.latency = 0;
   zeroCost.bandwidth = INFINITY;
   zeroCost.messageTime = 0;

   /* get graph containing all nodes of interest */
   remos_get_graph(nodes, &graph, &timeframe, NULL);

   nodes.remove(startHost);

   /* calculate times to send a message from the starting node to
      all others */
   costFromNode[0] = new Hashtable();
   costFromNode[0].put(startHost, zeroCost);
   compute_distance(node, costFromNode[0], length);
   clusterSize=1;

   while(clusterSize<clusterN){

     Remos_Node newHost;

     /* The array costFromNode has entries of how long it takes to
        get the message to every node in the graph from every node
        already in the cluster.  Go through the array and find the
        node not in the cluster with the lower average time to send
        the message. */
     newHost = pick_minimum_average_distance(costFromNode, nodes,
                                             clusterSize);
     /* add to cluster list */
     clusterNodes.add(newHost);

     nodes.remove(newHost);

     /* calculate time from the new node to all other nodes */
     costFromNode[clusterSize] = new Hashtable();
     costFromNode[clusterSize].put(newHost, zeroCost);
     compute_distances(newHost, costFromNode[clusterSize], length);

     clusterSize++;
  }

  return clusterNodes;
}
```

`compute_distances()` is used above to find the latency and bandwidth between a designated node and each of the other nodes in the network. Latency is the sum of latencies along a path and bandwidth is the minimum of the bandwidths of the links on the path.

```
 /* compute distances: return distance from one node to all others
  *          node: pointer to current node in traversal
  *     pathCosts: Hashtable mapping nodes -> NetworkCost, each entry
  *                holds the cost of reaching that node in the graph
```

```
 * messageLength: length of message to calculate time for
 *
 * This is a recursive routine that traverses the network topology
 * calculating the latency and bandwidth to each node.  The algorithm
 * is started by calling it with the desired source node and a
 * hashtable with an entry specifying 0 latency and infinite bandwidth
 * for that source node.  This algorithm will then traverse the graph,
 * adding up the latency and finding the bottleneck bandwidth to each
 * node.  pathCosts is used as a flag to indicate whether a node has
 * been visited before.  This algorithm is depth first, so it probably
 * won't produce good results on a graph with cycles.
 */

 void compute_distances(Remos_Graph_Node node, Hashtable pathCosts,
                        int messageLength){
   NetworkCost toGetHere = pathCosts.get(node);

   for(remos_list_all_elements(node->neighbors, neighbor_i, i)){
     NetworkCost costToNeighbor = pathCosts.get(neighbor_i);
     if(costToNeighbor == NULL){
costToNeighbor = new NetworkCost;
costToNeighbor.latency = neighbor_i->node->latency+toGetHere->latency;
costToNeighbor.bandwidth =
 min3(neighbor_i->node->bandwidth,
             toGetHere->bandwidth,
             neighbor_i->link->bandwidth);
costToNeighbor.messageTime = costToNeighbor.latency+
 messageLength/costToNeighbor.bandwidth;
       pathCosts.put(neighbor_i, costToNeighbor);

       compute_distances(neighbor_i.node, pathCosts, messageLength);
     }
   }
 }
```

2.9 Implementation

In designing the Remos API, we sought a design that is both portable, in terms of the applications it supports and the networks on which it will run, and implementable on current networking hardware. We believe that the API is generic and portable across a wide variety of network architectures, although the full information it seeks to provide may not always be available. We have also built an implementation of the API to demonstrate that it can be implemented on current networking hardware.

The Remos implementation consists of both a user- and system-level component. The user-level component is called the Modeler and provides the API to applications in the form of a library. The system-level component is named the Collector. At runtime, the Modeler receives queries from the application. It does some initial preprocessing, then passes the queries to the Collector. The Collector gathers the actual information from the network, then returns information about the network to the Modeler. The Modeler finishes processing the raw information into a response to the application's query, and returns the response to the user. The architecture is diagrammed in Figure 2.7.

The Modeler is currently implemented in both C and Java. It is single-threaded and communicates with the Collector over a TCP socket, using a simple ASCII protocol. The modeler is responsible for inserting virtual switches to simplify the topologies returned by the Collector and for performing max-min flow calculations on the Collector's topologies to determine solutions to flow queries.

If predictions are necessary, the Modeler uses Dinda's RPS package [35, 37, 38] to convert history information obtained through the Collector into a prediction of future performance. There are a number of issues associated with converting historical network information into future predictions, and these details are covered in Dinda's papers and in Chapter 3.

The Collector's architecture is more complex. Because Remos is designed to provide information about different parts of the network to a variety of applications requesting information, scalability was very important in the design. This prohibited having a single server act as Collector for the entire Internet. Instead, a two-layer scheme has been designed. The majority of the Collector design is described in a paper by Miller and Steenkiste [82]. I will summarize the design, as well as describe additional details that are being implemented and have not yet been published separately.

The first layer of the Collector design is the Master-Collector. The Master-Collector is primarily responsible for aggregation of the information provided by the lower-level architecture-specific Collectors into a single response to return to the Modeler. The Master-Collector keeps a database of which lower-level Collectors are monitoring which portions of the network. When a query arrives from a Modeler, the Master-Collector then divides the query into components corresponding to each portion of the network, as well as for the paths connecting those network portions. The components of the query are then sent to their respective Collectors. When the responses are received, the Master-Collector combines the information together again and returns it to the Modeler.

Each lower-level Collector is generally responsible either for one network or for the connection between networks monitored by other Collectors. There are three different types of collectors currently supported: SNMP Collectors, Bridge Collectors, and Benchmark Collectors.

The SNMP Collector is the basic Collector upon which Remos relies for most of its network information. SNMP is a database protocol designed for obtaining network-level information about topology and performance. Because it has direct access to the information the network itself stores, this Collector is capable of answering the flow and topology queries that require an understanding of the details of the network's structure. Details about how SNMP can be used to obtain topology and performance information is covered in later chapters.

The Bridge Collector acts as an assistant to the SNMP Collector. Its purpose is to provide topology information about a bridged Ethernet to the SNMP Collector when it receives queries for previously unknown nodes. The Bridge Collector queries all components of a

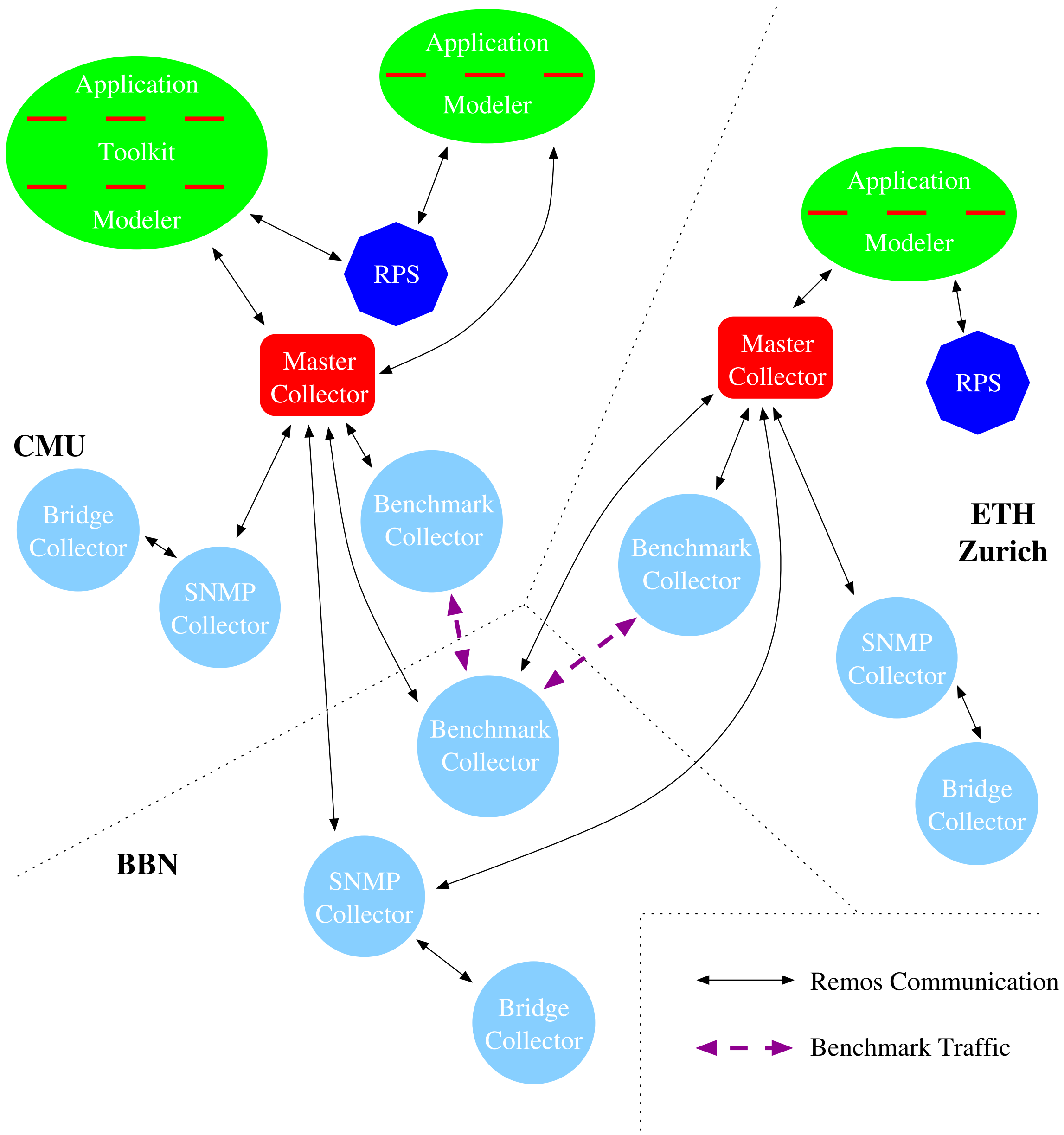


Figure 2.7: A detailed illustration of how the components of the Remos architecture are connected together. Shown here are applications running at CMU and ETH making use of resources at CMU, ETH, and BBN. CMU is at the top-left, ETH is at the top-right, and BBN is at the bottom. Each application is using prediction services to provide information about the future network availability. The applications at CMU are using machines at CMU and BBN, and the application at ETH is used machines at ETH and BBN. The benchmark measurements sent across the Internet are shown, but, for clarity, the connections between the SNMP and Bridge Collectors and the network components at each site are not shown.

bridged Ethernet to determine its topology, then stores this information in a database. When a query is made to the SNMP Collector, the Bridge Collector provides it with the portion of the topology that is needed to fulfill the query. Without the Bridge Collector, the SNMP Collector is only able to handle level 3 routed IP networks. With the Bridge Collector, it can provide information about both the level 2 and level 3 components of the network. The algorithms implemented by the Bridge Collector are described in Chapter 4.

While SNMP offers excellent information, it is generally not possible to obtain SNMP access to network information for WAN or other networks where you do not have an account on a machine. Because information is still needed to determine network characteristics connecting the networks where the SNMP Collector is used, the Benchmark Collector was designed. A Benchmark Collector is run at each site where an SNMP Collector is. When a measurement of performance between multiple sites is needed, the Benchmark Collector exchanges data with the Benchmark Collector running at the other site of interest. By measuring the rate at which the data travels across the network, the Benchmark Collectors determine the performance of the links connecting the network and report this information to the Master-Collector. Details and analysis of this implementation, in particular, is described by Miller and Steenkiste [82]. This technique is very similar to the techniques used by NWS for obtaining its measurements [119].

2.10 Related work

A number of resource management systems have been developed that either allow applications to make queries about the availability of resources, or to directly manage the execution of the applications. Systems in present use primarily deal with computation resources, such as the availability and load on the compute nodes of the network. An application may use this information to control its own execution or the system may place the application on nodes that have the minimum load. Examples of resource managers include research systems like Condor [72] and commercial products like LSF (Load Sharing Facility). While such systems can be adequate for compute-intensive applications, they are not suitable for applications that handle movement of large data sets and applications based on internetworking, since they do not include a notion of communication resources.

More recently, resource management systems have been designed for Internet-wide computing, some examples being Globus [47] and Legion [52]. These systems are large in scope and address a variety of mechanisms that are necessary to make large scale distributed computing possible. For example, Globus services include resource location and reservation, a communication interface, a unified resource information service, an authentication interface, and remote process creation mechanisms. The Remulac project and Remos interface are less ambitious, but more focused. We are concentrating on good abstractions for a network to export its knowledge to an application, and for applications to use this knowledge to achieve their performance goals. Thus, our research addresses some of the core problems that are critical to the development of large scale resource management systems, and we aim to develop a set of mechanisms and abstractions to allow the development of network-aware applications. Most network-aware applications require some policies to chose the appropriate mechanism. While these are beyond the scope of this project, they are the subject of other research efforts including the *Amaranth* project at Carnegie Mellon [55].

A number of groups have looked at the benefits of explicit feedback to simplify and speed up adaptation [39, 58]. However, the interfaces used in these efforts have been designed specifically for the scenarios being studied.

NWS is a system commonly used for obtaining measurements and predictions of network and system performance [117]. Its interface is designed less to ensure portability and more to expose the actual measurement numbers collected directly to the application. Because of its relationship to much of my work, the other aspects of NWS are discussed in subsequent chapters.

A number of program are collecting Internet traffic statistics, such as the NLANR passive monitoring project [85]. This information is not a in form that is usable for applications, and it is typically also at a coarser grain than applications would like to get. Another class of related research in the Internet is the collection and use of application specific performance data, e.g. a Web browser that collects information on what sites had good response times [98].

2.11 Summary

Remos allows network-aware applications to obtain information about their execution environment. Remos provides a uniform interface so that portable network-aware applications can be developed independently of any particular network architecture.

The challenges in defining the Remos interface are network heterogeneity, diversity in traffic requirements, variability of the information, and resource sharing in the network. The Remos API is the result of an effort to present the network at as high a level of abstraction as possible, while maintaining the important low-level information needed by many applications.

If an application needs only high-level information, it can use a flow-based query that provides information on the performance of application-level point-to-point flows. Because it has the low-level information about the network, Remos can account for the effects of sharing between different flows in the application. For applications that need lower-level information directly, the topology query primitive is provided. For the topology query, Remos returns a logical topology graph, which captures the information of interest to the application.

There are two questions to be answered when evaluating the Remos API. The first question is whether it meets the needs of real applications. Remos has been designed to support the various applications described in this dissertation and has been used by a several research groups for a variety of applications. Although presented at a high level, the abstractions presented by the Remos API provide the low-level information many applications need to properly optimize themselves to their network environment.

The second question is whether the information needs of the API can be met on the networks used by people running these applications. The remaining chapters address how low-level information about the network can be gathered and used to provide both low-level information, such as topology, as well as predictions of application-level end-to-end performance.

Chapter 3

Network-based Measurement

The development and performance of network-aware applications depend on the availability of accurate predictions of network resource properties. Obtaining this information directly from the network is a scalable solution that provides the accurate performance predictions and topology information needed for planning and adapting application behavior across a variety of networks. The performance predictions obtained directly from the network are as accurate as application-level benchmarks, but the network-based technique provides the added advantages of scalability and topology discovery.

The previous chapter discussed the need for a scalable means of providing predictions for distributed environments. The purpose of the network-based technique is to provide accurate predictions through a technique that is scalable enough to meet the challenges imposed by emerging distributed environments. That the network-based approach is scalable is not difficult to demonstrate—after describing and evaluating the technique, I will discuss its scalability. The more important questions concern whether the technique provides the same quality of predictions obtained through applying end-to-end benchmarking of the network. The majority of this chapter focuses on this evaluation. After verifying that the accuracy of the predictions is not sacrificed by using the network-based technique instead of end-to-end techniques, the scalability of the network-based technique and its implications for supporting grid-based computing will be discussed.

This chapter describes how to determine network properties directly from the network devices. It begins with a description of the general features required to support network-based measurement. It then provides an overview of SNMP and describes the features SNMP provides that make it possible to extract available bandwidth information from network devices. The available bandwidth predictions based on network queries using SNMP are compared with traditional predictions based on application history to demonstrate that they are equally useful. These results validate the use of low-level information to predict the application-level performance that is required in the network-based approach to performance prediction.

3.1 Network requirements

Before discussing how the prediction is done, it is necessary to consider what information is necessary for making useful measurements and predictions of network performance. This section first lists the requirements that a network must meet to support network-based measurement. Three popular network architectures: Ethernet, ATM, and Myrinet are analyzed in terms of these requirements.

3.1.1 Requirements

The requirements that the network components must meet are:

Intelligence

The first requirement is that the networking components, such as switches and routers, must support a protocol to communicate their status to other devices in the network. Commodity networking hardware uses SNMP, which is essentially a hierarchical database protocol, to serve this purpose. Devices supporting SNMP are frequently referred to as "intelligent." Except for extremely low-end bargain devices generally not found in commercial or academic environments, all networking components support SNMP.

The remainder of this chapter will simply assume that whatever information the network component provides is communicated using SNMP. SNMP was designed to allow network managers to remotely observe and adjust network components. It defines the structure of and operations on a database that is stored in each network component. The database is organized hierarchically, with portions reserved for various standards bodies and vendors. Components are free to implement only those portions of the hierarchy that are desired. Each portion of the hierarchy is specified by a document referred to as a Management Information Base (MIB). Although it is more correct to refer to only the database protocol as SNMP, in common usage SNMP is used to describe the collection of MIBs as well as the protocol. I follow common usage unless distinctions are needed for clarity. For more information about SNMP and MIBs, many books have been written for use by network managers [104]. Further details of the specific parts of SNMP that are used for network prediction are described in Section 3.1.2.

Characteristics

The networking component must be able to report the bandwidth of itself and the links to which it is attached. Latency is also important, although it may be more influenced by congestion than by the natural physical characteristics of the link itself. For the majority of network architectures, this information is simply the physical capacity of the links. If reservations or other policies for dividing the physical link across multiple virtual links are used, however, determining what the "capacity" of a link is may become more complex.

Utilization

The most important information that must be obtained from the device is the utilization of each link to which the device is attached. Currently, many network components do not directly report utilization. However, almost all components provide traffic counters that can be used to calculate a time-averaged utilization. Using counters, the frequency with which the counters are updated determines the intervals over which rates can be calculated.

The ideal design to provide utilization information for network components would involve the network device itself calculating utilization information. If the network component were to update statistics in real-time, it could provide useful information about burstiness and variability in network traffic that is difficult to capture by periodically sampling an SNMP traffic counter. Incorporating more useful statistics about traffic is one of the goals of the Apmmon proposal [33], an IETF proposal for standardizing performance statistics to be calculated by network devices and made available through SNMP.

Routing

Finally, the routing information must be available from the network. “Routing” here refers to whatever protocol is being run across the network, regardless of whether the messages are using IP or another protocol. For instance, ATM, Myrinet, and Ethernet each have their own routing protocol. The requirement is that it must be possible to determine the hop-by-hop path that a connection will take. This information may not be given directly, such as with Ethernet, but it must be obtainable.

3.1.2 Networking technologies

Ethernet

Ethernet, used for almost all LANs today, meets all of the requirements for providing network-based predictions. Being the most common LAN infrastructure, it is well supported by the basic SNMP MIBs. As originally developed, Ethernet was primarily bus-based, with bridges used to separate networks. Modern Ethernet networks use switches, which are bridges with many ports, to partition the network into separate switched segments. However, in most Ethernets the core of the network uses switches, while the endpoints may be connected to a hub, which provides a bus-based network for several endpoints. On bus-based portions of the network, each machine can observe all of the traffic on the network. However, on a switched network it is necessary to obtain utilization information from the switches themselves.

Information about Ethernet networks is obtained from two MIBs. The first is the standard MIB, which stores basic information for all network architectures. The second is the Bridge-MIB, which stores information used by Ethernet switches to forward packets around a switched Ethernet.

RFC1213 describes the standard MIB, called MIB-II [77]. It is intended to describe essential information needed for all network components—including hosts, routers, and bridges. It provides information about components' offered services and networking hardware. It also provides statistics and information about major networking protocols, including IP, TCP, UDP, and SNMP.

Three parts of this MIB are of interest for network-based measurement. The first is the interface table. The entry for each interface provides the maximum data rate as well as octet counters, which indicate the number of bytes the interface has sent and received. This table provides the link characteristics and utilization information needed for network-based measurement.

Another useful component of this MIB is the IP routing table, which indicates the address of the next hop used by the device to forward IP packets to their destination. This is the first item of importance in determining a network's topology. This table stores the IP routing only, which corresponds to the OSI model level 3 protocol. Once the address and interface used by the device to forward the packets has been determined, then the routing algorithm used by the level 2 architecture of that interface's network determines the next hop the packet will actually take to reach that next level 3 hop.

The final useful component of MIB-II is the address translation table, which stores the mapping from machines' level 3 IP addresses to level 2 Ethernet addresses. This information is what is returned to the UNIX command `arp`, and is needed for queries made to the bridges to determine the Ethernet's topology.

The second most important MIB is the BRIDGE-MIB [32]. This MIB provides information about the status of an Ethernet bridge, which is used to forward packets between different portions of a LAN. The interesting part of this MIB is the forwarding database, which stores the port used to reach each of the Ethernet addresses the bridge has seen. Because bridges operate transparently, making queries from this MIB on each bridge is the only way to obtain the information needed to construct the topology of an Ethernet LAN.

ATM

ATM networks can be more complex than Ethernet, primarily because ATM's early development focused on supporting new networking technologies such as virtual circuits and reservations. Elementary ATM networks require access to the IP-OVER-ATM MIB [71] and MIB-II. Other MIBs are used to describe some of the more advanced standard features available on ATM networks.

Myrinet

Myrinet [15] has quickly grown to be one of the best commercial hardware and software solution technologies for the development of scalable clusters. Myrinet employs switches that forward source-routed packets between nodes. The first generation Myrinet switches were dumb switches, providing no information about their status or utilization. Current Myrinet switches, however, provide capacity and utilization information. No routing information is provided through the switches. However, because Myrinet uses source-routed packets, routing information is already stored at all of the endpoints, and is determined automatically by a program distributed with the current generation of Myrinet software.

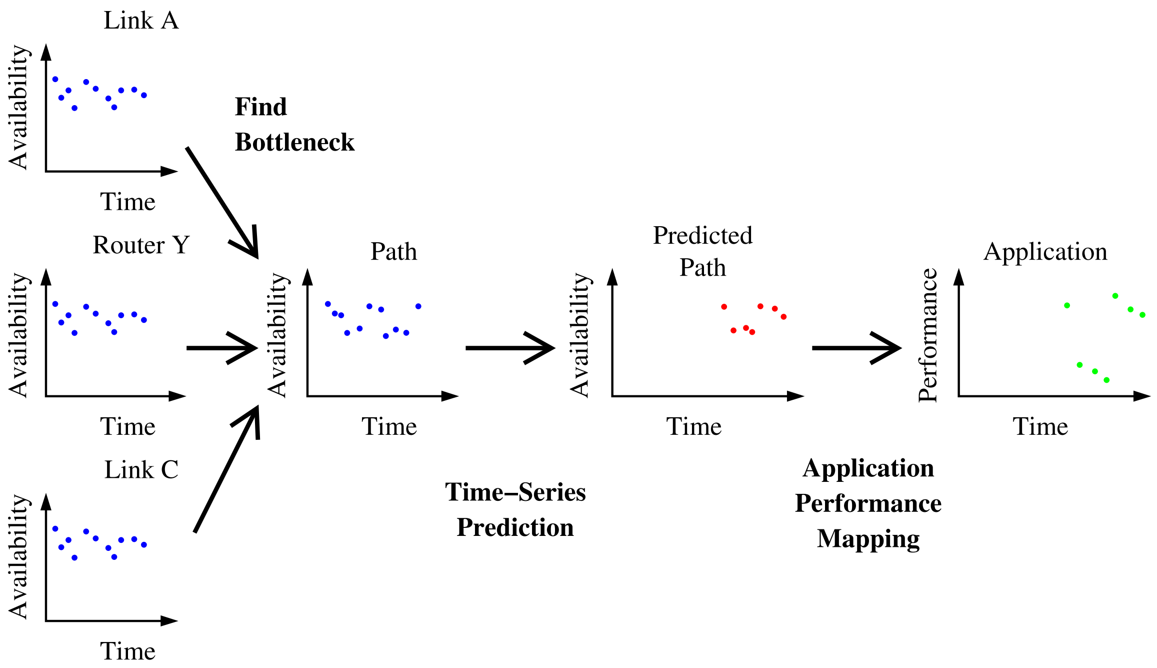


Figure 3.1: The stages involved in making a network-based prediction.

3.2 Network-based prediction

Figure 3.1 shows the process of making a network-based prediction. Each step will be discussed in detail.

3.2.1 Record a series of measurements

The first step in making a prediction is recording a series of measurements from each component in the network. As discussed above, Remos records these measurements by sampling traffic counters periodically using SNMP. Because a rate is needed, two successive samples are taken to calculate a time-averaged utilization for that time interval.

The sampling frequency used in this first step is the limiting factor for predictions made through this pipeline. More frequent samples will allow better understanding of the burstiness of the competing traffic on the network and allow for more accurate predictions to be made for the performance of fine-grained applications. Unfortunately, taking samples at higher frequency is more problematic, both due to the higher load imposed on the network and devices in taking the samples, and in terms of the inaccuracies in the counters and the sampling protocols being amplified as the sampling frequency is reduced.

A more ideal solution would be having the network device itself gather statistics on the traffic on each link. By doing this, the device could sample the counter at a much higher rate and collect information about burstiness and other characteristics that are more difficult to obtain using SNMP-based sampling. The Apmmon MIB proposal [33] is one proposal for an addition to SNMP supporting some of these monitoring techniques.

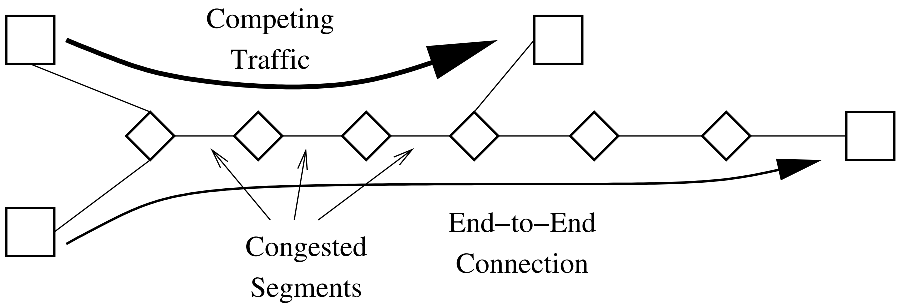


Figure 3.2: A path where correlated utilization observations will need to be accounted for.

3.2.2 Finding the bottleneck

The network-based prediction process for an end-to-end connection begins with separate series giving the availability of each step of the path between the two ends. These series of observations are stored independently until the prediction is requested.

To make a prediction, the first step is to calculate the bottleneck bandwidth between the two endpoints. End-to-end bandwidth is decided by whichever link has the minimum, or bottleneck, bandwidth along the path. For each time step in the series, the minimum bandwidth is selected from among the components along the path. A new series, reflecting the bottleneck bandwidth at each timestep, is then constructed for use in the rest of the prediction.

Initially, this step may seem unnecessary—it might be easier to simply maintain predictions of each component individually and only calculate the bottleneck when an end-to-end prediction is needed. Unfortunately, this technique ignores correlations that must be taken into account to produce accurate predictions. Consider the network shown in Figure 3.2. Here, the end-to-end connection requiring prediction will experience congestion from a single competing source along three different links. The congestion on all three segments has the same source, therefore they will be congested at the same time. However, if the future congestion on each segment is predicted separately, this correlation might be lost, and the prediction may turn out to be significantly inaccurate. In particular, if the three measurements are not synchronized, it is unlikely that a time-series model would predict the same future congestion on each segment. However, if the bottleneck is calculated first, then any regularity in the congestion will be detected and the prediction will correctly reflect the correlation of the congestion along the three segments.

3.2.3 Time-series prediction

The next step is to use the series of observations to predict the future utilization along the path. Time-series prediction is a general technique for performing such a prediction. A mathematical model is fit to the series of previous observations. That model is then used to obtain a prediction of the future behavior of the network. These models require uniformly periodic utilization samples and provide similar periodic predictions of future behavior.

One of the most important aspects of time-series models is their ability to provide both a prediction of the future value of the series, in this case the network's utilization, as well as a measure of the confidence in that prediction. That measure of the confidence in a prediction can be just as useful as the prediction itself. In particular, on networks with extremely invariant traffic, it will indicate that the predicted utilization will be almost constant for the application's run. On other networks with extremely bursty traffic, the variance may be orders of magnitude larger than the utilization prediction itself. This can occur on networks that are typically idle but occasionally have bursts of capacity traffic. Overall, the variance prediction is useful for indicating how well the network-based predictions can be trusted.

There are a large number of mathematical models for time-series prediction to choose between. Describing these and the significance of the different models is beyond the scope of this dissertation. For general information about time-series models, see the book by Box et al. [19]. Peter Dinda has completed an extensive survey of various options for time-series prediction and developed a software package that implements a number of these models that has been used for predictions in my research [34].

Rich Wolski compared the accuracy of a number of time-series models for making predictions of bandwidth for network benchmarks [116]. Wolski found that the sliding window and autoregressive models were both among the more accurate models available and inexpensive to fit to the series. For these reasons, these two models were chosen for the time-series prediction used in the following experiments. A more thorough technique would include evaluating which model best fits the current series and dynamically returning predictions using that model, but such techniques were not implemented in these experiments to simplify comparisons. RPS, the prediction package developed by Dinda, supports this technique [37].

3.2.4 Application mapping

At this point, the time-series prediction provides a series of predictions of the utilization of the network for the immediate future. The obvious question is how to convert this information into the application's actual performance. The simplest answer is to assume that the bandwidth received by the application will equal the capacity minus the utilization. Knowing the capacity and having a prediction of the utilization, simple arithmetic produces a result. However, the behavior of real networks and applications is much more complex than this simple equation.

Application behavior

The design and implementation of the application itself has a significant impact on the performance it ultimately receives from the network. Several factors may be of importance:

Granularity Is the algorithm fine-grained, with many small messages and responses, or coarse-grained, with fewer, but often larger, messages. A fine-grained application requires low-latency communication, with predictable bandwidth. A coarser-grained application may better tolerate variance in available bandwidth and deal better with high-latency networks. In particular, if the communication is planned so that the application can hide communication latency, it will behave better on many networks.

Communication software What message passing software is used? Some message passing implementations rely on multiple handshakes to ensure buffer capacity, others send data blindly, trusting on OS buffering and the application to handle the data. The efficiency with which the communication software makes use of the bandwidth available to it has a significant effect on the application's performance.

Communication protocol Does the application use TCP or UDP? TCP will attempt to use the network fairly, backing off it there are packet loss problems. UDP will not, of itself, back off. Depending on the circumstances, either protocol may achieve a higher bandwidth on a particular network. UDP may perform better in many cases, due to its lack of management overhead and its tendency to cause other TCP-based connections to back off, thus allowing it more bandwidth. On the other hand, in a situation where UDP data is suffering heavy losses, TCP's congestion-avoidance strategy may result in better throughput.

Competing traffic

Characterizing traffic by its utilization of the network is useful, but does not provide a complete picture of how that traffic will interact with the new application. The basic calculation of available bandwidth shown above simply assumes that the competing traffic will not be affected by the new application. This is not always true. On a LAN, the competing traffic may consist of a small number of TCP connections. Those TCP connections will react to the application's new connection and may actually provide the new application with more bandwidth than shown by the available bandwidth calculation. On the other hand, if the competing traffic consists of unreactive UDP connections, there may be no reaction, or even less bandwidth if the newer application induces loss in the existing connections and causes them to increase their data rate to compensate.

Similarly, the number of connections has an effect on the bandwidth an application may achieve. On a wide-area connection, there may be hundreds, if not thousands of simultaneous connections. One additional connection will have almost no effect on such connections, regardless of the protocols involved.

Although characterization of competing traffic is much more difficult to obtain than simple utilization measurements, the behaviors of different types of competing traffic can have a substantial impact on the performance received by new applications.

Network infrastructure

The hardware and software used to implement the network can have a significant effect on the behavior of the network. Beyond the obvious effects of additional bandwidth, the sharing policies used by different switches can change the bandwidth available to an application. Network policies designed to implement fair queueing, such as hashing different connections to different buckets, or random early drop, designed to force TCP connections to back off before causing loss, can improve the performance of a network segment and cause a network to behave dramatically different than a simple FCFS queueing policy.

Choosing a model

These factors leave several choices for a model for $A_n()$. The first approach is a general model. A general model would have to consider all of the characteristics of the application, competing traffic, and network hardware and software. Such a model is probably not feasible to develop and would certainly be too expensive for real-time prediction.

A simpler approach is to rely on past history for future predictions. By recording the performance of applications on a network under various traffic conditions, this history could be used to build a predictive model. Repeatedly monitoring the same application provides a characterization of the application's behavior. Similarly, monitoring the same network over time should result in an effective predictor of the types of traffic seen on that network. Although simple in concept, this approach requires large amounts of information to utilize effectively.

A third approach at developing the mapping would be to divide the applications and network traffic into categories, based on their characteristics. Rather than developing a single model to reflect all conditions, this allows less data to be used to make effective predictions. It requires the ability to classify applications and traffic according to their characteristics.

For the following experiments the second approach, based on previous history, will be used for predictions. This decision is primarily a matter of practicality. There are few portable techniques for determining application behavior, competing traffic characteristics, or network implementation. Rather than focusing on the development of new techniques for obtaining this information, I chose to determine whether the basic concepts of network-based prediction are fundamentally sound. Once network-based techniques are validated, further work can be done on extending the monitoring capabilities of the network, such as exploring options to obtain the information needed for more complex approaches.

Different implementations of the history-based prediction were used for the different experiments and are described individually below.

3.3 Testbed verification

While there are clear advantages to the network-based technique, making predictions about end-to-end operations using low-level information is inherently difficult [94]. I have verified the network-based technique against an application-based technique. If the two techniques offer similar accuracy, the scaling and efficiency advantages of the network-based method make it the better choice for performance prediction.

These experiments were performed on a dedicated testbed where the conditions could be controlled to represent a wide variety of congestion levels. Because of the breadth of conditions experienced on networks, it is important to test prediction techniques at all levels of congestion [91].

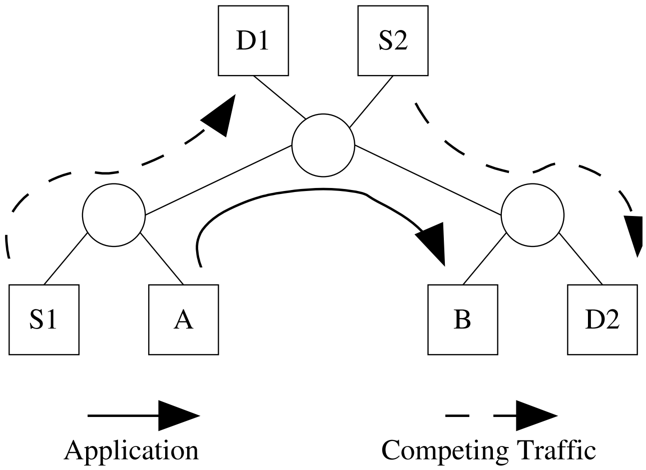


Figure 3.3: Topology of the testbed used for the prediction experiments. All links are 100Mb. The hosts are 300Mhz DEC Alphas and the routers are Cisco 7206 routers.

The network configuration used in the experiments is shown in Figure 3.3. The “application” used was a simple 1MB data transfer from A to B using TCP. Every 15 seconds, SNMP was used to measure the available bandwidth on all segments of the path between A and B, followed immediately by the application’s data transfer. To measure available bandwidth over different averaging periods, the SNMP traffic counts were obtained 5, 3, 0.5, and 0 seconds prior to the TCP message.

A 1MB data transfer would be a typical benchmark. However, for this experiment, it is considered an application because it is being used to predict its own, rather than other applications’, performance. A real application would also involve computation, which is being ignored for the purposes of this experiment.

Synthetic traffic was inserted onto the network between S1 and D1, and S2 and D2, resulting in two congested links competing for bandwidth with the application. This synthetic competing traffic was generated using fractional Gaussian noise (FGN), a method described by Paxson for representing realistic aggregate traffic encountered on networks [89]. The average rate of competing traffic on each link was chosen between 0Mbps and 100Mbps (link capacity) and changed an average of every 10 minutes.

3.3.1 Experimental method

A diagram of how the experimental data was processed is shown in Figure 3.4.

The first step of the process is to collect the experimental observations. As mentioned above, during the running of the experiments, traffic counts were obtained 5, 3, 0.5, and 0 seconds prior to sending the TCP message. The multiple datapoints were saved so that offline analysis could be done using different averaging windows for bandwidth measurements. For the results described in this section, only the half-second window was used. After the final traffic count was obtained, the TCP message was immediately sent. The pair

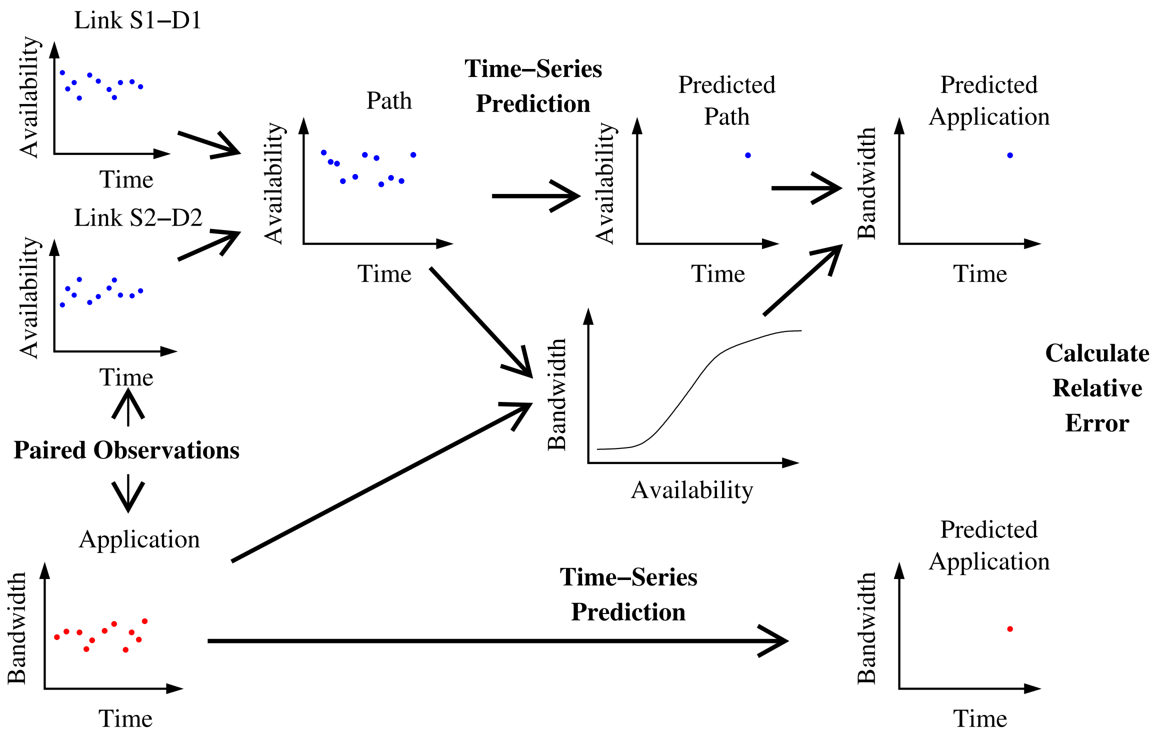


Figure 3.4: The process used for analyzing the data collected in the experiment.

of half-second traffic counts combined with the TCP bandwidth form the set of raw data used by the analysis for each time step.

The second step in the process is determining the bottleneck bandwidth along the path. Because there are only two links subject to congestion in this experiment, this step involves only selecting the minimum of the two traffic counts.

After the bottleneck bandwidth is calculated, the $A_n()$ mapping function can be built. $A_n()$ is a piecewise linear interpolation initially built for each data set with the training observations made before actual experimental predictions are attempted. As the experiment proceeds, each subsequent observation is added to the mapping function..

As discussed previously, the technique for building $A_n()$ is one of the most interesting design choices that must be made to apply network-based prediction. In these experiments, A_n is reevaluated each time. In a real system, this might prove too costly for real-time use if predictions for a large number of links are required frequently. My belief is that the calibration would be done far less often in a real system, but with more data than used in these experiments. For instance, if the type of traffic (TCP/UDP, number of connections, type of connections) on a link is characterized, it should be possible to apply one of a few known models to the observations of network utilization to predict the performance of the new application. However, such a project requires substantial research beyond the scope of my dissertation. I hope that my simpler approach demonstrates the validity of the basic techniques and encourages further research on the best models for predicting application performance based on network utilization measurements.

The application prediction was performed on the series of times recorded for the 1MB data transfer. Different time-series models were evaluated for predicting future application performance. The relative error between each step-ahead prediction, $A_t(\mathcal{N})$, and the next actual observation, $A(\mathcal{N})$, was recorded.

The network-based prediction was formed by applying the same time-series models used above to predict the step-ahead available bandwidth, N_t, using the series of bottleneck observations obtained in the experiment. The step ahead prediction was mapped to a prediction of application performance by applying the mapping function described above, resulting in the final prediction of $A_n(N_t)$. The relative error between each step ahead prediction and the next actual observation, $A(\mathcal{N})$, was recorded.

Due to the nature of network performance prediction, there is necessarily error in any prediction technique. The primary purpose of these experiments is the comparison of the relative error of the two techniques to determine if either technique is significantly more accurate than the other.

Experimental results

Over 65,000 observations were taken during the experiments. To determine the accuracy of the two prediction techniques, 30 sets of 1500 consecutive observations were chosen at random from the experiment. The first 1000 were used to fit the time series model. The prediction technique was then tested over the next 500 observations. The model was refit for each additional observation, so the time series model was only used to predict one observation interval ahead. Each prediction was compared with the next actual observation.

The implementation of the time series predictors that were used is described by Dinda and O'Hallaron [36]. The autoregression (AR) and sliding window average (SW) prediction models were used. Wolski examined several prediction models and found these two to be useful for network performance prediction [116].

A comparison between the relative errors is shown in Figure 3.5. The important observation is that there is little difference between the application- and network-based techniques. This leads us to conclude that the network-based prediction technique can be used to provide network predictions with accuracy comparable to application-based techniques.

It's interesting to note that in Figure 3.5 the results do not show that either the network-based or application-based technique are significantly better. The two best curves use both different measurement (network- and application-based) and prediction (AR32 and SW8) models. This combination of techniques indicates that the results do not show that one technique is significantly better than the other, just that there are many factors that affect the accuracy of a prediction. The observation that neither network- nor application-based is consistently better or worse is the most important, because it indicates that the network-based approach, although not as direct as running the application, provides equally useful information. Given the other advantages of the network-based approach in terms of scalability, invasiveness, and topology detection, it appears to be the clear preference from this experiment.

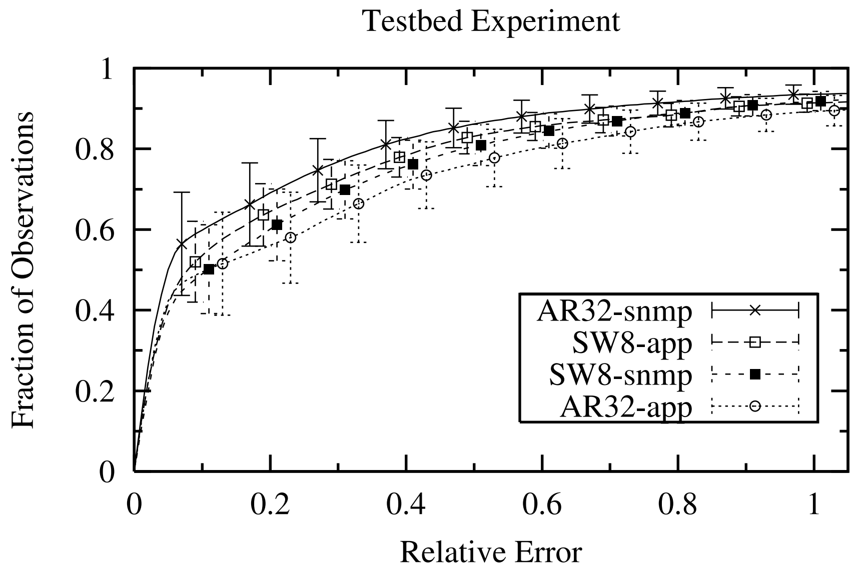


Figure 3.5: Cumulative relative error distributions for application- and network-based prediction. Both 32nd-order autoregressive (AR32) and 8 observation sliding window mean (SW8) predictions are shown. The SNMP available rate was averaged over 3 seconds for each observation.

Predictability of traffic

The analysis of these two prediction techniques relies only upon comparison of their accuracy with one another, rather than comparing them with perfect accuracy or an idealistic model. Other research has demonstrated that real network traffic varies widely in its predictability [7, 95, 116]. The most predictable traffic is generally aggregated traffic streams on Internet backbone links, as the aggregation forces the averaging of the dynamic behavior of many flows. On the other hand, LAN traffic, with sudden starts and stops of high-bandwidth applications, can be rather unpredictable due to the small number of start-stop applications that may be able to saturate the network links on the machines they are running. One approach used to deal with predicting future behavior is to make predictions as a range rather than as a specific value [34, 97].

Rather than focusing on a single type of traffic, the competing workloads in these experiments were chosen to create a wide range of competing traffic—from highly variable to rather stable congestion. In my analysis of the results, I observed no significant difference between the techniques under any of the traffic scenarios, therefore the results are combined into a small number of graphs.

3.4 Simulated verification

Because it is not possible to reproduce all ranges of network behavior on the testbed, simulation was used to explore a wider range of parameters. The NS simulator was combined with packet traces from a university network gateway to produce a simulation of traffic types not possible on the testbed.

3.4.1 NS simulator

The NS simulator is one of the most widely used network simulators available today [5]. This experiment used a limited subset of the NS simulator's capabilities. The simulation used a traffic trace to represent congestion on the network and a Reno TCP connection to represent the application's traffic.

3.4.2 Packet traces

The traces used were obtained from the Passive Measurement and Analysis project of the Network Analysis Infrastructure being developed by the National Laboratory for Applied Network Research [85]. This project has captured the headers of actual network traffic from sites across the Internet. The traces used in this experiment were captured at the San Diego Supercomputing Center's Internet commodity and VBNS connections.. The traces were collected during the weeks of July 19, 1999 and August 9, 1999.

The goal of this project was to collect traces of the network traffic without disturbing the traffic in any way. This was achieved using hardware to monitor the actual uplink connection used by major sites to connect to the Internet or VBNS. Dedicated PCs were used to monitor the traffic. Using an installed optical splitter and ATM interface, the machines were able to record the traffic on the link while remaining invisible to both ends of the connection. Collecting the data consisted merely of writing the network data to a high-speed disk. The final data could then be extracted from the actual traffic offline. The header collection hardware produces perfectly accurate traces, but buffer limitations restrict the length of each trace to 90 seconds.

For the simulation, the background traffic was generated as UDP packets replaying this tracefile. Because the traffic is replayed with UDP, it does not adapt to the network's congestion. This is a different behavior than would be seen in the original network environment. A more realistic combination of many TCP connections may be more appropriate, and possible to represent in the simulation. However, the modeling of user behavior, application response time, and network design affecting the behavior of the TCP flows are significant challenges and introduce their own additional modeling challenges. I have chosen to use realistic traffic traces in a somewhat unrealistic manner. Further study with additional traffic models is needed. As discussed above, the results obtained with these traffic models are promising and justify further research into the accuracy of these predictions.

Despite the limitations of this technique, it does represent several realistic behavioral characteristics of network traffic. In particular, when congestion is caused by the aggregated traffic of a large number of unrelated flows, a single connection will rarely have a

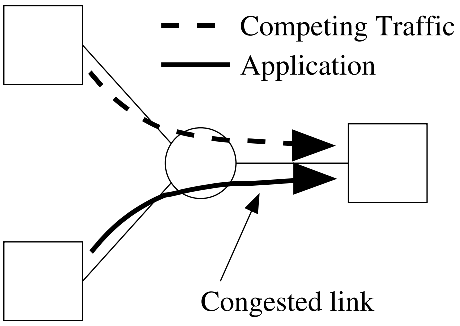


Figure 3.6: Simulated network topology. The marked link is the only bottleneck, resulting in congestion at the router that connects the three hosts.

significant impact on the competing flows. Because the traffic traces do represent a large number of flows (the entire campus of UCSD and SDSC during peak traffic times), the non-adaptive behavior is not as unrealistic as it might at first appear. Its invariance as the network topology is changed is somewhat less realistic, because the traffic generators would likely change behavior, but it is difficult to realistically predict user behavior in the face of additional congestion, and the congestion levels generated do appear in the modern Internet [91].

3.4.3 Simulated network

The network topology used for this simulation is shown in Figure 3.6. This structure was chosen so that the behavior of TCP under switch congestion could be studied. There are no bottlenecks at either the TCP or trace generating nodes or their connecting links. The two traffic streams feeding into the single congested link result in congestion at the bridge, which implements simple tail-drop queueing. Each trace was simulated with the bandwidth of the congested link set to 20, 30, and 40Mbps to provide different levels of congestion.

An advantage to running this experiment as a simulation is that the behavior of the TCP implementation could be changed slightly. When running a real experiment, it is necessary to consider the state of the TCP engine. In particular, shutting down an experiment early is difficult because both endpoints must agree that the connection is being closed. Until the link can be confirmed closed, the kernel maintains state for connections. Timing out this state may take several minutes. For the simulation, however, this complexity was unneeded. Instead, the TCP connection was simply left open for half a second. Rather than relying on the TCP connection to terminate normally, the simulation recorded the amount of data successfully acknowledged and reset both endpoints. This technique both removed the difficulty of dealing with improperly terminated connections and allowed shorter observations to be used for data gathering. It could be argued that one source of errors in the testbed experiment was that the length of time taken by the TCP connection varied tremendously based on the congestion it experienced during the transfer. By recording the amount of data transfered over a specific time period, this uncertainty was removed. A similar change is not easily accomplished with an actual operating system.

Because of the short length of the traces, each observation consisted of $1/2$ second to take the "SNMP" measurement and $1/2$ second for the TCP connection, followed by a $1/2$ second pause before the next observation began. The simulated switches do not support SNMP, but they do log their traffic counts at the required points, providing the same information that would be obtained from real equipment using SNMP. Over 10000 observations were taken using the short data sets. For each tracefile, the time series models were trained for the first 8 to 16 observations, and the prediction quality measured on the remainder.

$A_t(\mathcal{N})$ was built using the series of times recorded for the $1/2$ second TCP transfers.

The SNMP-based predictions were done by applying the time series models to the series of trace-based "SNMP" available bandwidth measurements. For each data set, $A_n()$ was created using the other data sets.

3.4.4 Results for heavily congested networks

The simulator allowed the analysis of performance prediction under much heavier traffic loads than the testbed experiment. As a link grows more congested, the available bandwidth reported by SNMP approaches zero. The available bandwidth, however, does not indicate the offered load, which is the amount of data applications are attempting to send through the link. For the same low available bandwidth measurement, the offered load producing that low reading may range from almost the link's bandwidth to orders of magnitude higher.

If the link is only lightly congested, a path for which SNMP reports little available bandwidth may actually provide an application with a higher rate than the amount available. This behavior can occur if the competing traffic's rate is reduced in response to the new application's traffic.

If the offered load is significantly higher than the link's bandwidth, it will be hard to get any data through. The router preceding the congested link will be dropping many packets already, and the competing traffic will be just as quick as the new application to use any bandwidth that becomes available.

It is interesting to note that the type of congestion seen in LAN and WAN scenarios tends to be different. When congestion is observed in the LAN environment, it is typically due to a small number of high bandwidth flows. In a WAN environment, congestion is more typically caused by an extremely large number of low bandwidth flows. In the LAN environment, a new application flow will cause the other flows to back off and they will generally share the total link bandwidth evenly. In a WAN, a new application flow will have little effect on the existing flows. Generally, each additional application flow will receive approximately the same bandwidth in a WAN. This difference is important, but more information than utilization is needed to predict the differing behaviors. Currently, there is no standard way to determine the number of competing flows on a link.

There is a difference between the way in which the synthetic traffic used in this experiment reacts to this environment and how real-world traffic behaves. Because the traffic in this experiment is non-reactive, it does not adapt to the packets being dropped due to congestion. While this behavior is unrealistic, many parts of the Internet do experience severe congestion. As many TCP flows attempt to share the same bottleneck, their continual

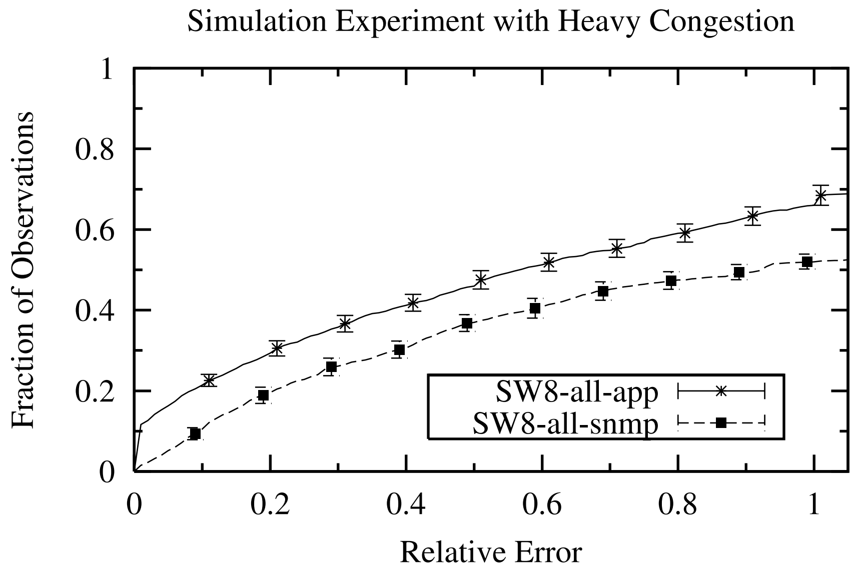


Figure 3.7: Relative error distribution with 90% confidence intervals for the simulation experiment including observations during extremely heavy congestion. For each technique, the relative error is calculated for the step ahead prediction versus the next actual observation. The results from all the traces were distributed among 30 data sets.

attempts to increase their bandwidth will result in packets being dropped at the congested point of the network. Therefore, because heavily congested links with dropped packets do occur in real networks, it is important to analyze the predictability of such networks. However, because the synthetic traffic is not adapting to the network, it may be causing dropped packets in very different ways than actual adaptive traffic might. In particular, in real networks the rate of dropped packets is related to the actual bottleneck bandwidth. In this simulation, there is no such relationship. In summary, the predictability of network traffic in congested networks is important. While some of the results of this simulation may provide useful information about the behavior of network-based prediction on congested networks, complete analysis and validation of the technique will require more complex scenarios realistically emulating network behavior under congestion.

The behavior of the application on a heavily congested network should be represented in $A_n()$ and is dependent on the type of network and competing traffic involved. For a congested network, information beyond that available through octet counters may be obtained through the count of dropped packets, which is also available through SNMP.

My initial analysis of the data revealed an interesting effect of the heavy congestion generated by some traces. The data shown in Figure 3.7 was surprising because there is a significant difference between the two prediction techniques and because the accuracy of both techniques is less than seen in most other experiments. Analyzing the data revealed that the majority of the errors occurred when the network was under extremely heavy congestion and that the network-based technique was more prone to these errors than the application-based technique. This discrepancy is due to the network-based technique's inability to measure the offered load on a congested link, whereas the application-based technique provides a history indicating whether the link is only marginally congested or seriously overloaded. Because these errors only appeared under extreme congestion and because such scenarios are of little interest to most distributed computing applications, observations where less than 1% of the link's bandwidth was available were removed from the data used in Figure 3.8. The inclusion of dropped packet information would likely address these shortcomings, but because my research has focused more on locating and avoiding networks under such heavy congestion than on precisely predicting their behavior, this functionality has not been added to the prediction software.

3.4.5 Results for moderate congestion

Figure 3.8 shows the aggregate results from all of the simulations. The simulation results also confirm that the accuracies of the application- and network-based techniques are very similar. It is interesting to note that in the simulation results, the results group according to the time series model chosen, whereas in the testbed results in Figure 3.5, the best results from each technique were with different time series models. This merely serves to illustrate that it is very important to select the most appropriate model. Prediction systems such as RPS [36] compute several time series models and report results from the one with the lowest error. Therefore, it is probably most appropriate to consider only the top curves from each technique.

3.5 Statistical metrics

The similarity in the results from the network-based and application-based techniques is very promising, because it indicates that a more efficient technique offers the same accuracy as the established technique for measuring network performance. The confidence intervals in Figures 3.5 and 3.8 indicate that both techniques have similar variability, but the differences between the testbed and simulation results require some explanation.

The results from the testbed experiment were divided into separate independent data sets. Because the background traffic was changing continuously, each data set was taken under different network conditions. This accounts for much of the variability in Figure 3.5—it is much easier to make accurate predictions on a lightly congested network versus a heavily congested network. For the simulation results, however, the individual tracefiles obtained were too short to produce enough data for a CDF. Instead, the distributions were created by randomly dividing the data amongst 30 sets. This randomization homogenized the data and removed much of the variability from the final result.

The CDF plots of the results tend to obscure the results for both low and high relative errors, in the first case because the distribution rises quickly and in the second because the distribution is long-tailed. To compare the two techniques over the entire distribution, Figure 3.9 presents a quantile-quantile plot for the simulation experiment. For two identical distributions, the quantile-quantile plot will produce the line $y = x$. In this graph, the two

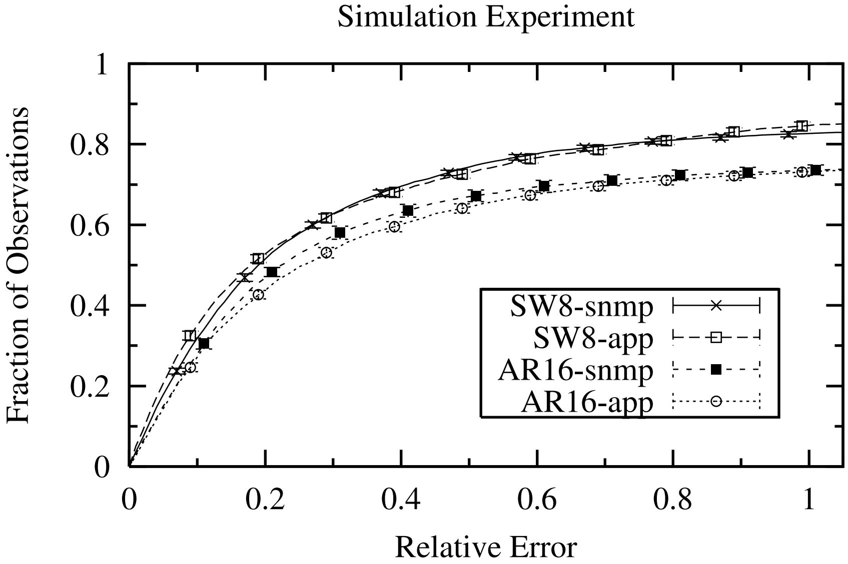


Figure 3.8: Cumulative relative error distributions with 90% confidence intervals for application- and network-based prediction in the simulation experiment. Both 16th-order autoregressive (AR16) and 8 observation sliding window mean (SW8) predictions are shown. The small error bars are discussed in Section 3.5.

distributions are almost perfectly linear, indicating that the distributions are nearly identical.

3.6 Scalability

The scalability of an algorithm can be defined in many different ways. For parallel computing, scalability generally refers to an algorithm's ability to achieve more or faster work through the addition of more processors. To a certain extent, any of the measurement techniques discussed in this dissertation are scalable—they can be run on very large numbers of machines. However, the amount of information that can be provided about larger networks and the applications that can be supported by that information varies depending on the characteristics of the measurement technique employed.

The scalability of a measurement and prediction technique is defined in terms of the number of machines that it can support. Scalability is, therefore, dependent on the support requirements of the applications that desire to make use of those machines. As such, it is impossible to broadly quantify the general usefulness of any of the available measurement techniques without first defining the environment in which they are being used. I will begin by discussing the considerations for scalability of the measurement techniques.

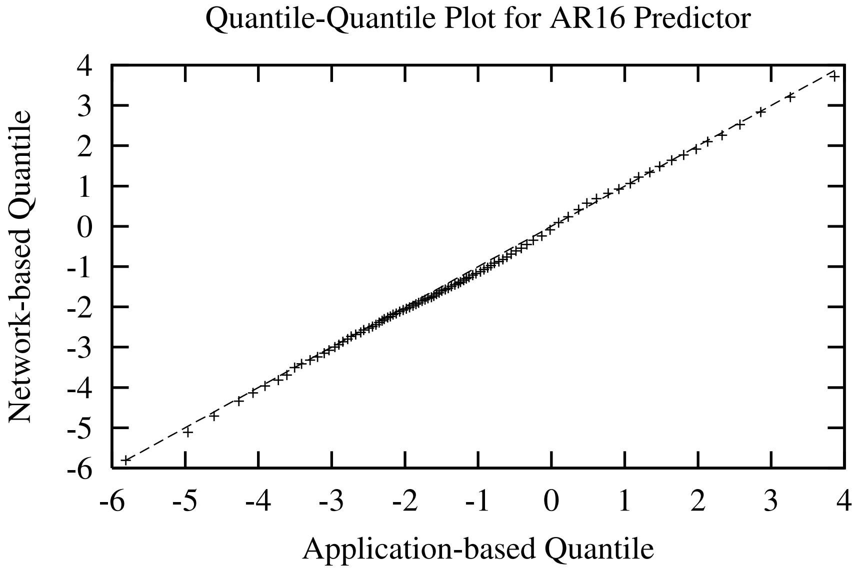


Figure 3.9: Quantile-quantile plot of the CDFs for each prediction technique. This is a plot of the logarithms of the values so that the similarity of the two techniques in both the low and high relative error regions can be seen. The $y = x$ line is shown for reference.

3.6.1 Scalability requirements

The two factors that determine scalability are completeness and frequency. For an individual application, completeness is only important insofar as the machines being used, and the connections between them, are being monitored and predicted accurately. However, rarely is the environment that needs to be monitored this simple. First, consider a single application that begins by selecting the set of processors it wants to use. The application needs information about all possible machines it might use and the connections between them. This may be a significantly larger number of machines then will actually be used.

An additional complication is brought about by the desire to provide monitoring for environments where many applications may be running. In fact, to achieve environments such as a computational grid, we must be able to provide information for any application seeking sources of computational or other resources. The very nature of this environment requires monitoring the network's performance between all of the machines that might be chosen for execution by any application. This challenge, in particular, is only met by a monitoring solution that is capable of spanning the entire network of interest.

The second component of the scalability analysis is the frequency at which the measurements are taken. The frequency of measurements is important because it determines the granularity at which predictions can be made. On a gross scale, the difference can be between one observation per day and one per second. Obviously, the second observation frequency is more useful than the first. But this comparison brings up the important question, which is determining what the best observation frequency is. Any of the techniques under discussion can monitor a network of any size, given a low enough observation frequency. Therefore, the scalability question is only defined when posed in the context of a minimum frequency requirement to support the classes of application intended for execution in that particular environment.

To evaluate the extent to which completeness and frequency need to be supported by a measurement technique, consider the requirements of three common steps of applications running on distributed systems.

Processor selection

First, consider the general processor selection process. Most distributed applications begin by selecting the processors to use during execution. For some applications, there may be some restrictions, such as the location of a particular resource such as an instrument or data storage, or a processor with a particular capability. Even those applications may have considerable flexibility in the placement of much of their computational work. The spectrum ranges up to applications that have no restraints on where they are placed. For the purposes of this example, I will consider only initial task placement at the start of execution, deferring the question of dynamic placement for later.

To make a processor selection decision, both the computational power of each machine and the communication capacity between the nodes needs to be taken into account. For this dissertation, I ignore the computational needs of the applications and focus only on their communication requirements. Applications have a variety of different communication styles, each of which dictates different specific requirements, but a few broad observations can be made about most applications. I will assume that an application has enough communication requirements to affect its performance, as embarrassingly parallel programs are of little interest here. The broadest question that an application can ask is what the network performance it will get over its runtime will be. That itself is an interesting question. At first glance, an answer of “6Mbps” might seem appropriate. However, if that number is only an average, it may not be particularly informative. If the 6Mbps result is achieved on a network that oscillates between 1 and 11 Mbps available, the network will not be nearly as usable. In particular, many applications are synchronized in some way—if a single message between two processors is held up, it will delay the entire application. Even if 99 of 100 links the application is using are free, one bottleneck will slow the entire application down. Similarly, many applications alternate between computation and communication, communicating for a small amount of their execution time. These applications are impacted much less by the average bandwidth available during their execution than by the bandwidth they are able to utilize during the precise portion of execution during which they are trying to communicate. These cycles can be very short and are affected by variability in available bandwidth over hundredths of a second. Not all applications are affected by such temporary fluctuations. A bag of tasks application, with little synchronization, might not be particularly affected by fluctuations, as it would only impact the time a single processor spends acquiring its next task or returning results from its previous task. These examples illustrate the importance of matching the frequency of observation with the needs of the application.

While my analysis of processor selection is extremely general in nature, I have depicted why the problem is more challenging than simply picking a set of machines with a high average available bandwidth. In order to effectively predict its performance on a network, an application must have information about all portions of the network that it considers for use. Therefore, we conclude that completeness is very important for processor selection. Furthermore, the frequency of observations necessary for prediction will vary between applications, but there are large classes of applications which require rather frequent measurements.

Load balancing

Next, consider the process of load balancing a running application. Load balancing can have multiple meanings, from shifting a running application between processors to redistributing data among the already utilized set of processors. The first definition is essentially the processor selection problem again, the important differences being that the cost of moving the data must be taken into account, and the load the the application itself is imposing on the network must be taken into account in determining how much bandwidth will be available for the application after redistributing its load. I will be discussing the later definition, where the application is already running on a set of processors and a performance improvement is desired by redistributing the data amongst these processors.

Load balancing is generally performed when running a synchronized data-parallel application. Although it is most commonly done because of uneven computational power between the nodes, other factors can be involved in the decision. Some applications have an irregular workload, meaning that although the data may be evenly distributed, the work might not be. If an application is on a heterogeneous network, it may be that a particular node does not have as high a bandwidth to its communication partners as other nodes used by the application have with their partners. This node should have less data because its communication phase takes longer to complete. Whatever the motivation, the decision is made by determining the current rate at which the application is running, a (higher) rate that can be obtained by redistributing the data, and the cost of moving the data.

The cost of moving the data is another short-term cost, requiring only brief predictions. The rate at which the application will run is again dependent on the type of application and the length of its communication phase. This computation is dependent on the same factors as predicting the speed of the application for the original processor selection. Again, if the application is synchronized, its speed may depend on the variance of network performance on the time-scale of the communication phase.

Dynamic scheduling

I will conclude by discussing a dynamically scheduled application. The Dv project at CMU has focused on developing a distributed visualization toolkit for supporting the visualization of large datasets over a distributed environment. The principal motivation for this project was the Quake project. The Quake project developed extremely large earthquake simulations, which in turn produce very large datasets. These datasets are large enough that it is not feasible to transfer them over a network, nor is running X-based visualization an effective solution for transmitting high-resolution interactive video. To address these difficulties, the Dv project has developed a system for dynamically scheduling the visualization of such datasets across the network, with the goal of providing a high-resolution interactive visualization at the user's desktop, in environments ranging in both network bandwidth and computational power.

The heart of the Dv system is the active frames used to deliver the images to the user's desktop. Each active frame is given the data needed to produce the frame and a deadline when it should be displayed on the user's desktop. The frame must then be dynamically scheduled to obtain the computational work and network bandwidth it needs to be processed and delivered before its deadline. Scheduling the Dv frames is a fascinating challenge, because there are several different components of the frame pipeline, each requiring different amounts of work and producing different volumes of data. Depending on the network and computational environment, it may be beneficial to perform most of the work at the computer center where the Quake dataset is stored, or to transfer the dataset to the user's network environment for processing. The array of options for execution and the different environments that Dv targets leads to a large number of possibilities for execution. However, the key component is the scheduling of the individual frames. Because each frame is short-lived, the scheduling decisions are made off of a very short timeframe—a typical frame may have 5 seconds of time before it must be displayed. If it needs initial processing at the remote site, transmission to the remote site, and additional processing before rendering, this indicates a need for forecasts of both computational and networking resources at the second (or finer) granularity. Peter Dinda has addressed the problem of predicting computational capabilities in this environment [34], I use this as an example of the need for high frequencies of measurement in the networking environment.

Lessons from the applications

While I have discussed only one specific application, I have discussed application scheduling procedures which are used by a large number of applications. Two conclusions can be made by considering these application requirements.

Individual applications may require only a small number of machines to be monitored. However, if a number of different applications are run in the environment, and in particular if distributed applications that may use many machines across the environment are used, then the monitoring system must be capable of monitoring all the machines in the network.

The frequency at which the network must be monitored is, again, dependent on the types of applications that may be using the network. Many applications, however, do require a measure of the variance in network performance on a fairly fine level. Although this frequency may be even finer, I will state that a one-second granularity should be sufficient to meet the needs of most applications. The monitoring and prediction of network traffic at a finer level should prove to be a challenging research project.

Based on the examples discussed above, the scalability requirement for a measurement system is that the system must be able to provide measurements of all paths in the network on a second granularity.

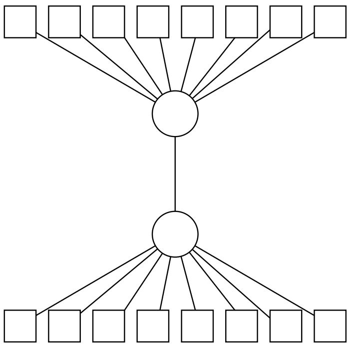


Figure 3.10: A simple way to connect 16 machines with 100Mb Ethernet.

3.6.2 Benchmark scalability

I will analyze the scalability of network-based and benchmark-based network measurement through analysis of how they perform on the network shown in Figure 3.10. This network is not the ideal way to connect 16 machines together using 100Mb Ethernet, which would generally be done using a single switch. However, it is common for larger networks to have subsets that can be represented by such a network. Furthermore, not every network is ideally designed, and real-world network measurement tools must be be capable of providing information about such networks.

Benchmark-based measurement

There are three factors that determine how benchmarking will be done on a network.

- Pattern: The pattern used is the first decision. The simplest pattern is where each endpoint performs a benchmark with every other endpoint, resulting in the predictable $O(N^2)$ number of connections.

  The most common means of reducing this cost is to perform the benchmarks in a hierarchical fashion, performing $O(N^2)$ benchmarks within a department, then between departments on a campus, then between campuses. This approach is of little use in a network such as Figure 3.10, even though it does have a hierarchical arrangement. Furthermore, not all hierarchical networks can be measured accurately in this manner. For instance, if the middle link had a capacity of 155Mbps, no single benchmark would be able to gauge its availability.

  One extreme example of reducing the number of benchmarks needed to measure every component in a network is to perform benchmarks between carefully selected pairs of machines, then algebraically determine the congestion on each link based on the connections that use the link. The first fundamental question is whether the topology is even known, so that the variables (bandwidth on links) can be determined. Using a pure benchmark-based approach, this is generally not possible. Finally, there is the question of performing independent benchmarks. Every path in this network involves at least two links. Algebraically, the minimum number of measurements that must be done in this network is 17. However, there is a problem with this approach—it is impossible to take simultaneous measurements. This prevents the use of algebra to determine the available bandwidth on links, because there is no guarantee that successive measurements across a link saw the same traffic conditions. Furthermore, because only bottleneck links significantly effect bandwidth, these measurements would not necessarily reveal information about links less congested than the bottlenecks.

As a result of the complexities associated with other schemes, measurement within a LAN is typically done using the $O(N^2)$ algorithm, with various synchronization techniques employed to prevent simultaneous measurements congesting a single link.

- Data length: Choosing the amount of data sent as a benchmark is equally important. Too little data can cause inaccurate readings. In particular, TCP slowstart prevents TCP from achieving its full rate during the beginning of a data exchange. While this is not a significant problem on slower connections, particularly on WAN connections, it can be a problem on LANs, where links typically have high available bandwidth. Alternative techniques, such as packet trains, require very little data to take an accurate measurement, but sample only an extremely brief moment in time, requiring more measurements to establish a better gauge of variability.

  Increasing the amount of data used for the measurement may produce a more accurate measurement on high-bandwidth network. Too much data may be harmful, however, as a long-lived connection for measurement may average too much time to be useful—in fact, variable time length measurements are an issue of some concern to the statistical techniques used for the measurements. Fixed time-length measurements of variable data length are more useful, but harder to implement.

- Load factor: The most serious concern about the amount of data sent by the benchmarks is the load question. How much of the network's bandwidth is consumed by the benchmarks being performed? If benchmarking is used prior to program execution, this may not be a significant concern, but if measurements are needed throughout execution because it is a shared or dynamically scheduled environment, then the amount of bandwidth consumed by the measurements themselves can be a significant concern.

  Again, consider the network shown in Figure 3.10. Here, the link between the switches is on the path used by half of the measurements. Unless the benchmarking is carefully planned, far too much of this link's capacity may be used simply by the benchmarking.

  Avoiding excess loading may be difficult using benchmarks alone. In fact, without network-based information or external expertise to provide the topology, the only safe assumption that a measurement routine can make is to assume that all paths in the network use the same links, requiring them to avoid sending any more data than would congest the bottleneck link in the network by the targeted load factor.

Obviously the scalability of benchmarking algorithms is highly dependent on the performance factors chosen. The best way to calculate the scalability of benchmarking on this network is to select appropriate parameters and determine the frequency at which the benchmarks can run. As discussed above, in most cases the only feasible communication pattern is N^2, so that pattern will be used. The data length will be selected at 100K, which is still not long enough to accurately measure the available bandwidth of the 100Mb Ethernet, but is somewhat more accurate than 64K, a standard parameter selected for such benchmarks [118]. The load target will be 10% of the network's capacity. Starting off with the assumption that nothing is known about topology, the algorithm will be restricted to generating an average of no more than 10Mbps. After accounting for overhead, this amounts to 1MBps of actual data. Using these figures, and assuming perfect utilization and synchronization between the processors, measurements can be taken every 25.6 seconds.

Arguments can be made regarding precise choice of parameters for this environment, but even if the topology is known and a 64K message is used, the frequency can only be worked down to once every 8 seconds.

For many applications, this measurement rate is more than sufficient. However, for others, such as dynamically scheduled applications, it may not be frequent enough. In particular, if an algorithm's performance is affected by variations that occur from second-to-second, it provides little information about the performance or variability of the network over such a time scale.

While many applications' needs are met by the information provided by this algorithm, consider further the scalability implications. The network in Figure 3.10 is extraordinarily small, and can only be measured every 8 seconds. Consider a more reasonable sized network with 200 nodes in it. Using the original algorithm, measurements are now restricted to less than one per hour. Clearly, the scalability limitations inherent in benchmarking prevent this information from being useful to many applications.

There are a number of modifications to these techniques that can be pursued. Attempts at deriving topology information can be made or optimistic assumptions can be made that allow multiple measurements to be made simultaneously. I will not attempt to evaluate these techniques, as each technique has its own benefits and drawbacks. Rather, I will simply discuss how the network-based prediction solves these problems.

Network-based measurement

In short, network-based measurements are scalable because they have an extremely small cost that grows with $O(N)$. Rather than making N^2 measurements for Figure 3.10, network based measurement requires $N + 2$ measurements. Taking all of these measurements consumes less than 18K of bandwidth and consumes an almost immeasurable fraction of a modern switch's processing power. These measurements can trivially be taken several times a second. Furthermore, while the number of measurements required to monitor a larger network grows, multiple machines can be used to take those measurements, allowing for the taking of measurements to be scaled.

3.6.3 Practical implementation

While the scalability of network-based prediction makes it a promising technique for many environments, there are still limitations that prevent its deployment as the sole source of network measurements for many environments. Foremost among these restrictions is the requirement that access be available to the switches and routers that form the network. In general, this is possible with minimal difficulty in local or campus environments. However, for wide-area networks, direct access to the network devices is typically not available. In these environments, it is necessary to use a hybrid of the network-based and benchmark-based measurement schemes.

The hybrid approach is currently implemented in Remos. Furthermore, it is also very similar to the architecture of the Network Weather Service. In NWS, benchmarks are performed within the LAN at each campus, and then between each campus site. One of the reasons this scheme is successful is that, in general, the bandwidth and latency bottlenecks across the wide area network are much more severe than in the LAN. Therefore, the assumption that wide-area traffic can ignore bottlenecks in the local area is almost always true. This same architecture used by the NWS can also be used to integrate network-based measurements directly into systems such as NWS.

Beyond issues of device access, there are additional issues regarding network measurement that may influence the choice between benchmarks-based and network-based prediction. The experiments I have performed to validate the network-based prediction approach have been based exclusively on network utilization information. Unsurprisingly, as shown by my simulation experiments, this approach falters when used on a saturated network, where complete utilization is the normal condition. While expansion of the network-based technique to incorporating dropped-packet information may alleviate these problems, further research must be done to validate expanding the approach. However, several of the arguments that work against benchmark-based measurement in terms of scalability no longer apply to congested wide-area networks. For example, load factor is no longer relevant, because the bandwidth achieved by the benchmark will represent a miniscule fraction of the total bandwidth—and the inclusion of one additional benchmark has almost no effect on the bandwidth achieved by subsequent applications. Furthermore, because of the aggregation of thousands of connections, the performance of the network has much less short-term variability. These practical issues make the hybrid network-based for LANs/benchmark-based for WANs scheme a very powerful technique.

3.7 Related work

A variety of systems have been developed to provide network status information. Two examples that provide applications with benchmark-based predictions are NWS [117] and Prophet [115]. SPAND [98] records similar data by storing applications' actual performance during execution and making this data available to help future applications.

All prediction systems described here utilize time series prediction techniques [19]. Wolski has studied several different time series models for their usefulness in predicting network performance [116].

SNMP has much broader uses than those that are described here. It is used to control and monitor a wide range of network resource properties [104]. Busby has explored using SNMP to gather information about both network and CPU resource as an addition to NWS [22].

Although SNMP provides much of the information needed for distributed computing, it is difficult to get it in the form required. For example, there are traffic counters for each port, but determining a traffic rate requires multiple, carefully timed queries. It would be much more appropriate to have the router or switch calculate its own time-averaged rate. Preliminary work toward this goal is discussed in the Apmmon Internet-Draft [33].

Recent network research has focused on modeling the self-similarity in network traffic, and these models may lead to more realistic traffic than Poisson processes. We used fractional Gaussian noise to generate self-similar traffic for our experiments [89]. More realistic wavelet models have been investigated more recently [93], however these techniques still suffer from the problem that they generate non-adaptive traffic. Further experiments on network-based performance prediction will more likely benefit from adaptive synthetic traffic.

3.8 Conclusions

My experiments have demonstrated that the accuracy of the network-based technique is equivalent to the accuracy of the benchmark-based technique. Their performance with very predictable and with less predictable competing traffic is very similar. While my implementation of the network-based approach falters when faced with a link with extremely heavy loss, its performance should improve if dropped packet counters are included in the information used for prediction. The similarities in the techniques' accuracies indicates that other criteria, such as scalability and invasiveness, should be considered when choosing a network measurement and prediction technique.

The results in this chapter demonstrate that it is possible to provide accurate predictions of application-level performance using low-level information. The prediction techniques described in this chapter utilize a history of application performance that is used to build the predictive model from network status to application performance. Making the jump to full deployment will require a different approach to building the prediction model that was used in these experiments. The calculations for building these models would need to be done less often, and would not have the application history for every observation as was available in these experiments. However, due to the accumulation of applications' history over time and an understanding of the characteristics of particular networks and applications, the frequency at which such calibrations are required should be lower, allowing accurate models to be built. In summary, the initial results are quite promising, but further research is needed to deploy network-based prediction in production environments.

Chapter 4

Topology Discovery

Knowledge of network topology is essential for application mapping. Without it, a scheduler cannot predict the links that will be shared by different messages of the same application. This sharing will have a large effect on the performance of the application. Topology knowledge also simplifies scheduling algorithms by making it possible to schedule in a hierarchical fashion, rather than analyzing all combinations of machines. Finally, network-based performance prediction requires network topology knowledge to determine which network components are involved in the path about which queries are being made.

Topology discovery is difficult because user-transparency has been a great driving force behind the success of networking. As a result, there are no common protocols for determining topology. However, the necessary information can be extracted using SNMP. I have developed algorithms to derive the topology with the information available through SNMP from much of today's networking hardware. My algorithm for topology discovery of a bridged Ethernet is the first such algorithm with provably good performance when using incomplete knowledge.

When analyzing topology discovery algorithms, the most important metric is completeness. Because topologies typically change infrequently, rapid topology calculation is relatively unimportant. However, if an application is predicting its performance on the network, and one of the nodes it is using is either not placed on the topology or is placed incorrectly, then the performance prediction will be severely compromised. Therefore, I focus on the completeness of the topology discovery rather than the rate at which the topology can be determined. In practice, all of the algorithms described here calculate the topology of even large networks within a few minutes of obtaining the information from the network devices.

4.1 Network structure

Figure 4.1(a) shows the network view that is presented to the user and that is preserved by most programming libraries. The IP routers connecting these machines are revealed in Figure 4.1(b). These are the easiest components to detect. In most cases, the traceroute program can be used to detect routers between hosts.

Figure 4.1(c) exposes the second level of transparency. This layer consists of the bridges that form the Ethernet LANs connecting the machines, and it is the most difficult level of

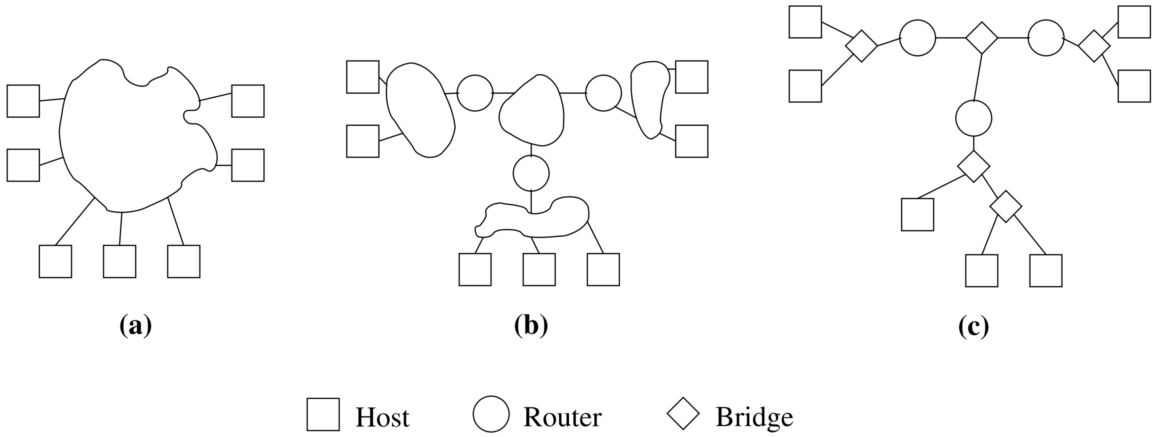


Figure 4.1: A view of networking at different levels of detail. (a) The view presented to the user. (b) The view at the IP routing layer, where each host and router explicitly forwards packets to the next component in the path. (c) The view including Ethernet bridges, where each bridge learns where the hosts and routers are and transparently forwards the packets towards their destinations.

topology to penetrate, even though Ethernet is the most common LAN infrastructure. The difficulty comes from the beauty of the transparent bridging protocol. The algorithms that the bridges use to determine how to form the LAN and how to forward packets require no global knowledge, nor do the hosts talk directly to the bridges or are aware of their presence in any way [92]. Thus, the goal of transparency is completely met, at the expense of the ease of determining the topology of the network. It should be noted that modern networks are typically built with "switches," which are essentially bridges with many ports. The terms "bridge" and "switch" are used interchangeably both in common practice and in this dissertation.

Despite the difficulties, Remos must locate all components of the network topology before using network-based measurement on that network. Figure 4.2 shows an example where available bandwidth predictions will be useless because a congested link occurs between two undiscovered bridges.

4.1.1 IP routing

IP routing topology is relatively easy to determine because the routing table each host and router stores and reports via SNMP explicitly lists the next hop on the route used to reach each destination. The IP routing topology may therefore be determined by following the routers hop-to-hop from source to destination.

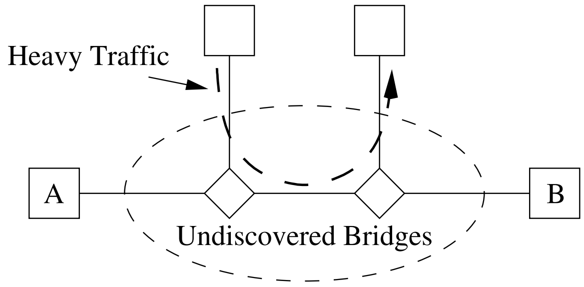


Figure 4.2: An example of a network configuration where missing bridges can produce misleading results. In this case, missing the two bridges between A and B misses the congested network link between them and may vastly overestimate the available bandwidth.

4.1.2 Bridged Ethernet

As originally designed, Ethernet LANs consisted primarily of shared 10Mbps segments. In large networks, bridges could be used to tie shared segments together, frequently just to tie two networks together. As Ethernet has evolved, bridges became cheaper and are now the primary means of interconnecting machines. Shared segments are limited to a handful of machines. These segments are connected with switches, which are bridges used to connect many shared segments together. By utilizing switches rather than shared segments, the aggregate bandwidth of the network is increased by separating communication that can follow different paths. 100Mb, 1Gb, and aggregated links allow portions of the network supporting heavy traffic to have a higher bandwidth than given to the endpoints, forming a fat-tree.

Bridged Ethernet networks are required to use an acyclic, or tree, topology. The acyclicity requirement simplifies the task of forwarding packets by ensuring that there is only one path to any machine on the network. If switches are connected to form a cycle, they will use the Ethernet spanning tree algorithm to deactivate one link of the cycle, thereby ensuring the network remains acyclic.[1]

The Ethernet bridging algorithm is much more complex than the IP routing algorithm. A bridge learns how to forward packets by listening to all traffic sent on the links attached to its ports. Whenever the bridge sees a packet on any of its ports, it stores the packet's source address and the port on which that packet was seen. This information tells the bridge the port on which the source machine can be found and will be used when the bridge later receives a packet that is destined for the source of the current packet.

While the bridge builds its forwarding database (FDB) by observing the source address of each packet, it must also forward each packet it receives to the port on which the destination node is found. At the same time it learns from the source address, the bridge must select from three options for forwarding that packet to one of its ports:

[1]The spanning tree algorithm can be disabled on most bridges, but if a cycle is created with links, the network will no longer function properly.

- If the destination machine is found on the same port the packet was received on, do nothing.
- If the destination machine is found on a different port than the packet was received on, the packet is resent, unchanged, on the port for the destination machine.
- If no port is known for the destination machine, then the packet is "flooded," meaning it is sent to all ports of the bridge, except for the port that received the packet. Hopefully the destination machine will reply to the packet's sender. As soon as the bridge sees the response packet, with the source address set to the previously unknown machine, it will place the appropriate entry in its FDB so further flooding will be unneeded.

This algorithm is known as transparent bridging, which is currently used on Ethernet LANs almost exclusively. More information can be found in Perlman's book [92].

Because this algorithm is completely transparent to the hosts, it is difficult for anyone to find bridges automatically or even to determine their names. One solution is for the person running the discovery code to obtain a list of bridges from an external source, such as the local network manager. A more general solution is to scan the LAN for bridges. This can be accomplished by sending an SNMP query for an entry in the forwarding database of the Bridge MIB to all IP addresses belonging to the local IP subnet. If the device responds with a forwarding entry, then it is in use as a bridge on the network. If it does not respond at all, or gives a response other than a forwarding database entry, it is either not willing to speak SNMP with the querying node, and therefore not useful in the topology search anyway, or it is not in use as a bridge on the network.

For bridged Ethernets supporting multiple IP subnets, each range of addresses must be checked separately.

Before beginning a bridge topology discovery program, the routing topology must be determined, as in Figure 4.1(b). Then the bridge discovery algorithm is run in each subnet between the routers to determine the bridging topology.

4.2 Effects of topology on applications

The topology of a network is one of the most important factors in determining the performance an application will see. The previous chapter dealt exclusively with predicting the performance achieved by an application sending a single message between two points. This is an extremely important topic. However, relatively few parallel applications send data only between two points in a network. Almost all parallel applications are designed to exchange several messages simultaneously. Even applications that utilize a single master processor for scheduling purposes frequently have multiple simultaneous communications occurring on the network.

The end-to-end performance predictions discussed in the previous chapter address the simple case of sending one message. The performance of multiple messages may be independent, if none of the messages interfere with each other, or the part of the network they share is more than adequate for their combined bandwidth. Alternatively, they may

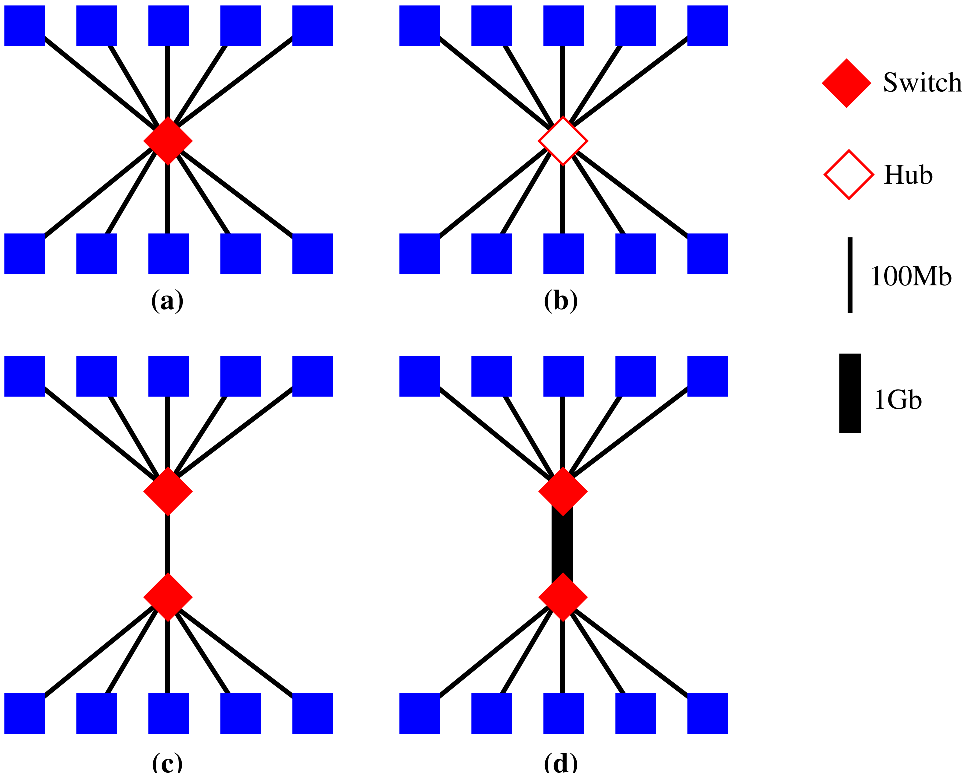


Figure 4.3: Four networks used to connect ten machine, each of which produces 10Mb for a single measurement, but whose performance varies widely when used in parallel.

share the same bottleneck, drastically reducing the bandwidth they receive by a factor of approximately the number of messages sharing the link.[2] The network topology must be known to predict how a parallel application sending simultaneous messages will perform on a real network.

Consider the networks shown in Figure 4.3. Each of these networks represents a different way to connect ten machines together. What is unusual about these four networks is that each of them provides exactly the same bandwidth between any individual pair of nodes (in the presence of no other traffic). However, the performance of the networks varies drastically depending on the choice of network and combination of machines being used.

Although simple, each network in Figure 4.3 represents a very realistic problem that is regularly encountered on real networks. The obvious solution for building a high performance 10 processor network is to use the single switch as in Figure 4.3(a). Unfortunately, the reality is that due to naiveness, cost constraints, or simple evolution as upgrades are made over time, non-ideal systems are here to stay. Furthermore, each of these networks might only be a component of a larger network. Although 128 port switches are available,

[2]Note that there is a third case, where messages apparently sent simultaneously in the code are not actually sent simultaneously during execution. However, the questions of application scheduling and synchronization are beyond the scope of my dissertation.

only a network specially purchased for such a large environment would ever use only a single switch. In keeping with the principles of opportunistic distributed computing, my approach has been to enable applications to make use of what power is available to them, rather than restricting myself to only well-designed systems.

The pure switched network in Figure 4.3(a) represents the ideal network configuration. Almost all modern switches have the internal bandwidth to switch their full, or near their full, traffic load between all ports. To the endpoints, the switches essentially appear to be a crossbar, with congestion observed only if multiple messages are sent or received by a single node. It is almost always best to build a dedicated cluster around a single switch. Thanks to the increasing size of switches, this is an achievable goal in many situations.

Figure 4.3(b) represents the bargain basement approach to network design. No one would knowingly design a high-performance computing cluster connected with a hub. Unfortunately, the rise of commodity computing has also allowed people who don't really understand the technology to build such clusters. The very low cost of current hubs makes them an attractive purchase to bargain shoppers. Additionally, hubs appear in other less-than-ideal environments. In the CMU CS department, and many others, what was originally a shared Ethernet was divided between floors with a single switch. Gradually switches were pushed further from the center of the network towards offices. A switch was placed on each floor and groups of offices were assigned to each port on the switch. Today, most offices are assigned to a dedicated switch port, with a hub connecting machines in each office. The wiring in the building has been extended in anticipation of providing each machine with its own switch port.

Concurrent with the evolution of the general department network has been the development of a number of independent research networks. Although primarily used in project labs, a few different networks have spanned the building, resulting in some machines in offices being connected to a totally different network than other machines in the same offices. Because these research networks are controlled by other groups than our normal networking staff, they are maintained to different standards than the central network. For instance, although the departmental network has focused on providing switches wherever possible, there is a research network that has 40 machines on a hub-connected shared segment. Even in our "modern" environment, hubs exist where they shouldn't.

Figure 4.3(c) shows a more typical network, where each machine is connected to its own port on a switch, but the two switches are connected with a link that can become a bottleneck when used by multiple simultaneous messages. Because no single pair of machines can detect this bottleneck, it is important to have a network-based technique to acquire the topology to determine such bottlenecks.

Figure 4.3(d) represents an improvement in the common network configuration depicted in Figure 4.3(c). Rather than connecting the two switches with a link equal to the endpoint links, a higher bandwidth link is used. Gigabit Ethernet has made this type of connection much more feasible—in fact, our current departmental per-floor bridges are connected with the central bridge using two aggregated gigabit Ethernet links. Prior to gigabit Ethernet, 155Mbps ATM was commonly used to supply higher backbone bandwidth, although it was easily congested with multiple 100Mbps clients.

In Figure 4.3(d), this backbone link eliminates the possibility of congestion on this network. However, were more machines connected to this network, it would again be possible to generate congestion. In this case, it is impossible to detect such a link without using a very large number of machines.

4.2.1 Benchmark-based topology

Because of the importance of topology information for scheduling distributed applications, a number of projects have studied ways to analyze the topology of networks, both wide-area and local-area networks. Several such projects are described below.

WAN topology

Wide-area performance is fairly easy to measure and it is much easier to predict because of the effects of traffic aggregation. There are frequently very significant performance differences between pairs of machines among several sites. Additionally, for most network environments the WAN bandwidth is at least an order of magnitude lower than LAN bandwidth. This difference allows benchmarks to be used to measure wide-area performance with little concern about the influence of local-area characteristics on the measurements.

The most common technique for determining WAN topology is to use benchmarks to measure bandwidth across the topology. This technique is rather trivial to implement when running measurements from a single machine to a variety of other Internet sites to calculate their distances from the first site, but more recent projects have focused on using measurements from multiple sites to build a topological picture of the Internet. Topology-d was an earlier project that used measurements to build a minimal spanning tree view of the network [87]. Current work, such as the IDMaps project [61] and the work of Theilmann and Rothermel [112] build network distance maps, a more flexible representation of the network's topology. Their research has studied the best types of benchmarks as well as placement of the sites running the benchmarks for accurate topology determination.

A lower level approach has been to use the hop-by-hop feedback provided by tools such as traceroute. The Mercator project [51] has explored this technique to group IP addresses by network topology produce an Internet map. Although many modern routers no longer respond to traceroute packets, they have achieved good results on the modern Internet. NIMI [1], which is a general architecture for controlling Internet probes, has been used to control both bandwidth and traceroute benchmarks and has also provided useful information about the separation between Internet sites. Skitter has been developed by CAIDA to combine traceroute and benchmark-based analysis [23]. Octopus combined SNMP routing information, traceroute, measurements, and heuristics to determine network topology [101].

These projects have all been quite successful at providing information to applications concerned about the impact of wide area topology. However, because of the performance and implementation differences between WANs and LANs, they do not address the issue of LAN topology discovery.

LAN topology

Although there has been a significant amount of research studying the importance of topology in task placement for local area networks [2, 30, 44, 62, 69, 103], there has been less work on the automatic determination of LAN topology than WAN topology. A number of projects have looked at discovering the topology between IP routers, but because the most interesting portions of LAN topology are generally formed by level 2 devices, they have been unable to address the majority of LAN topology issues.

My earlier work with ECO addressed the LAN topology problem by sending benchmark messages between every pair of machines in the network [73]. At that time, ECO was able to use application-level measurements to detect the performance penalty imposed by the switches and routers used to connect machines in the CMU CS department. Those measurements allowed ECO to build an accurate picture of the network's topology. Fortunately, network technology has advanced to the point where observing the impact of network devices is no longer that simple.

More recent work by Shao, Berman, and Wolski has focused on using measurements benchmark measurements to determined functional differences between machines in the network, which they refer to as effective network views [99]. Their technique has three stages. First, they send a benchmark between a central test machine and each other machine of interest. The initial results are used to partition the machines into clusters with similar bandwidths to the test machine. The next step is to perform the same measurement simultaneously between the test machine and pairs of machines in each cluster. If the measurement performance degrades significantly, then the machines are kept in the same cluster, if not they are placed in separate clusters. After completion of this phase, the machines in the network are divided among clusters between which there is a noticeable difference in network performance with the test machine. A third phase may be used between machines in a cluster to detect private networks within that cluster.

Although there is the allowance for private networks, effective network views focus on the performance of the network as seen from a single machine. From this perspective, it offers a very complete view of the network's performance and offers information regardless of whether the machines are distributed across a LAN or WAN. For an application such as the master-slave Mandelbrot set application studied in their paper, this view is ideal. On the other hand, there are environments and applications for which the single machine's perspective of the network is less than ideal. Applications without a single master exchanging messages, or applications wishing to select the best master, will benefit from views of network performance from different machines in the network.

4.2.2 Motivation for explicit topology

It is easiest to motivate the need for topology information when discussing a regular application with large amounts of collective communication—because many messages will be exchanged simultaneously, the penalty for messages sharing links can be very great. To show that this information is also useful with irregular applications with independent tasks, I will instead begin by analyzing the QuakeViz application according to its topology requirements.

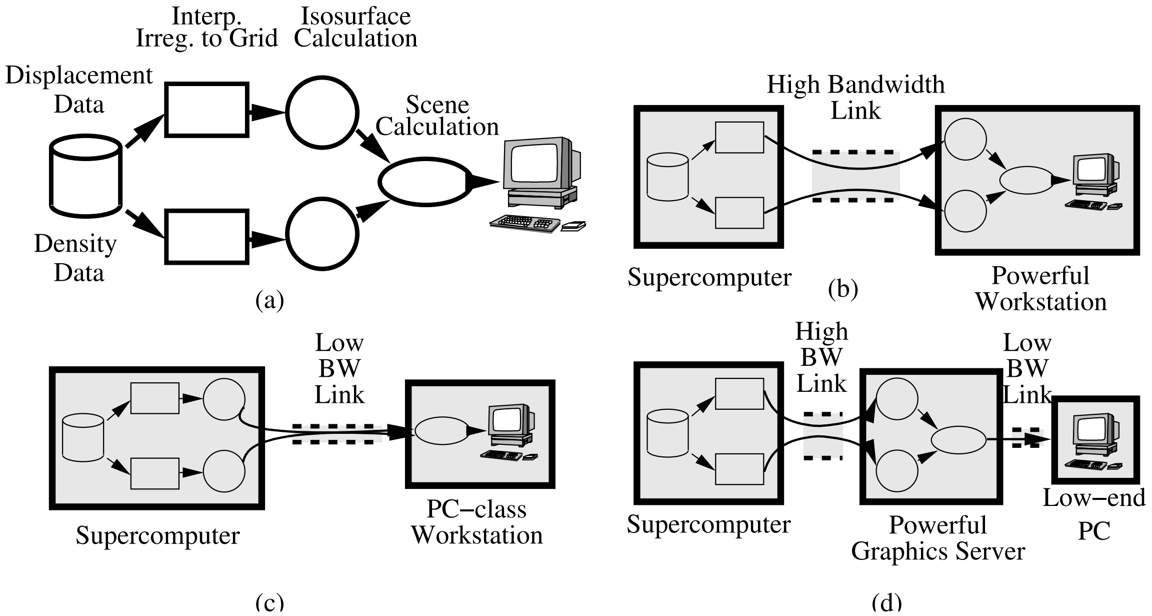


Figure 4.4: The stages of processing a frame for QuakeViz. a) shows the stages needed for processing, b), c), and d) show various partitionings of the process, according to the resources available to the applications

QuakeViz was previously discussed in Section 3.6.1. The application consists of pipelined tasks, with varying communication and computation costs between each phase. Depending on the network and computational power available, it is possible to perform the entire application on a remote server, or to do some processing at the remote server before sending the partially processed data to a machine closer to the client for final finishing. Because QuakeViz is a real-time application with performance requirements for interactive use, it is important that the resources used for the computation be capable of meeting the application's needs in real time.

Figure 4.4 depicts several possibilities for partitioning the phases of the QuakeViz application on different processors according to the available resources. The portion of the problem that is not illustrated by these drawings is that, because of the time required to render each frame, there are several QuakeViz frames being processed simultaneously. Therefore, this is not merely a single pipeline, but a parallel pipeline, as depicted in Figure 4.5.

The difference between conventional pipelining and parallel pipelining has several implications for scheduling QuakeViz applications. The most important is that although the application is conceptually a pipeline, because it is parallelized multiple frames will be sharing the system resources simultaneously. Although the active frames used to implement QuakeViz are each dynamically scheduled, the parallel nature of the frames needs to be considered, as well. The links along the path between the client and server must be able to handle the load generated by the simultaneous frames. Furthermore, various parameters that affect the workload for each frame are chosen when the frame is created and require additional cost to change later in the execution process. If the frame is ini-

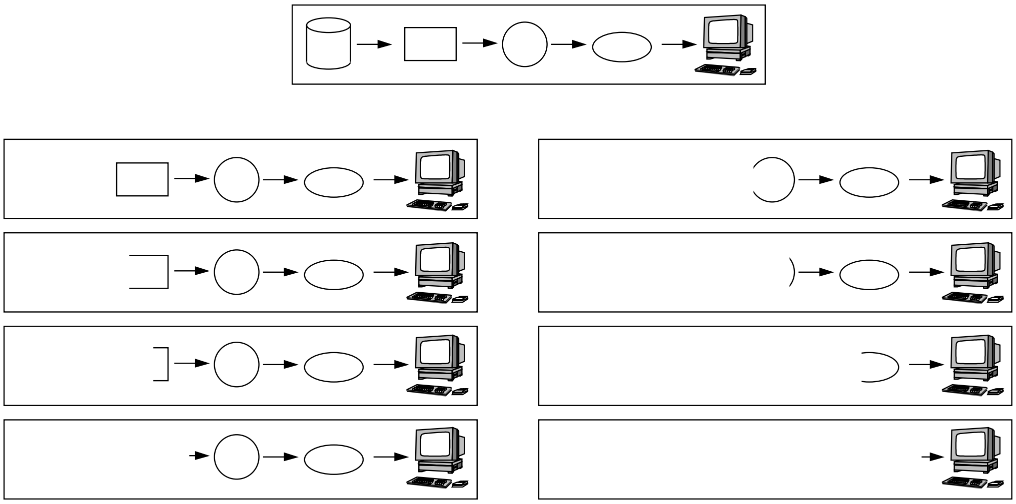


Figure 4.5: This figure illustrates how the parallel pipeline approach to a distributed application results in several frames at varying stages of completion being processed simultaneously. The top diagram indicates the phases each frame completes, with the lower diagrams indicating the progress of each frame. The scheduler must ensure that each frame receives the computation it needs and can be transferred across the network to the client desktop prior to its deadline.

tially planned to be sent over to the client's site for final processing, but upon reaching that phase, insufficient computational power is available, then the frame will either need to be changed, or processed remotely and transferred to the client's location. These changes may correspondingly affect the network load imposed between the sites.

The central issue in scheduling QuakeViz is understanding the relationship between the workload involved in planning the application and the location of the computational resources along the network topology. Locating the computational resources needed by the application on the network's topology graph allows the link sharing between the parallel frames to be considered in planning the execution. Although dynamically scheduled, there is a substantial penalty for a frame missing its deadline. If a frame is completed, but cannot be delivered, or must be transferred back across a congested link because a mistake was made in where to schedule the frame, application performance will suffer. Without knowledge of the network's topology, it is possible to make substantial mistakes that can prevent reasonable real-time interaction from being achieved.

4.3 Processor selection example

Processor selection is one of the key issues for distributed applications. Although I briefly discussed it above, its importance merits a further look. One of the first uses of the Remos interface was a processor selection algorithm used by several applications. Without the support of the Remos topology interface, it would have been difficult, or even impossible, to develop. In this section, I will include a description of the algorithm and the results it obtained to demonstrate how a real program made use of the topology information available in Remos. More details may be obtained from the original paper [107].

4.3.1 Node selection procedure

This algorithm makes use of the topology graph available through Remos, as well as additional information on computational power. For notation purposes, the network topology graph is an undirected connected graph $G(n)$ containing n nodes. A node in this graph is a *compute node* or a *network node*. A compute node represents a processor that is available for computation, while a network node represents a network device used for routing communication. The edges in the graph represent communication links between the nodes.

A function $cpu(i)$ is defined for each compute node n_i and it represents the fraction of the computation power of the node that is available to an application. The cpu function is computed from the load average on a processor as follows:

$$cpu = 1/(1 + loadaverage)$$

The justification is that the load average represents the number of active processes, and the processor will be equally shared by those processes and the user application process. That is, we are implicitly assuming that all jobs have equal execution priority.

Functions $maxbw$ and bw are defined for each pair of nodes connected by an edge. $maxbw(i, j)$ is the peak bandwidth between the nodes n_i and n_j, while $bw(i, j)$ is the corresponding currently available bandwidth. We also define $bwfactor$, as the fraction of the peak bandwidth that is available, i.e.:

$$bwfactor = bw/maxbw$$

Fundamental node selection algorithms

We present algorithms for node selection for maximizing available computation capacity, for maximizing available communication capacity, and with different weightage to computation and communication. We make a number of assumptions here, in part to simplify the presentation. We assume that the network topology graph is acyclic, i.e., there is only one path for data to travel between a pair of nodes. We also assume that the network environment is homogeneous, i.e., all computation nodes are identical and all communication links have the same capacity. The basis used for communication optimization is only bandwidth. These assumptions as well as well as the algorithmic changes required to relax them are discussed in the full paper [107].

Maximize computation capacity

For a homogeneous system, node selection for maximizing available computation capacity can be done effectively by simply choosing the nodes with the least amount of load. In our graph terminology, if an application requires m nodes, we simply select the m nodes with the highest values of the cpu function.

Input: A connected logical topology graph $G(n)$. Number of nodes required for execution m and given that number of compute nodes in G is at least m.
Output: A set M containing m nodes that maximizes the minimum bandwidth between any pair of selected nodes.

1. M = $\{$Any m compute nodes in $G\}$
2. Remove the edge with the minimum available bandwidth (lowest *bw*) from G.
3. Find the largest number l of connected compute nodes in G, and let the corresponding connected graph component be L.
4. If $(l > m)$
   M = $\{$Any m compute nodes in $L\}$
   Goto Step 2.
5. M contains an optimal set of m nodes.

Figure 4.6: Algorithm to select a set of nodes to maximize the minimum available bandwidth between any pair of nodes

Maximize communication capacity

The criterion used for node selection for maximizing available communication capacity is to *maximize the minimum of the available bandwidth between any pair of selected nodes*, i.e., to minimize the bottleneck communication path. Note that we are assuming that all communication links have equal capacity, but the *available bandwidth* on a link is a dynamically varying quantity, and will, in general, be different for different links. An algorithm that selects a set of nodes to optimize available communication capacity based on the above criterion is outlined in Figure 4.6.

This algorithm is based on the following simple observation. For a set of connected nodes in an acyclic topology graph, the least bandwidth between any pair of nodes in the set cannot be less than the lowest edge bandwidth in the graph. Hence, by repeatedly removing the minimum available bandwidth edge until there no longer exists any connected component with m nodes, the node-set that maximizes the minimum available bandwidth between any pair of nodes is obtained. In terms of the algorithm description in Figure 4.6, the size l of the largest connected component L of graph G will keep decreasing as the minimum bandwidth edges are removed from the graph. Eventually this size will become less than m, the number of connected nodes required for execution. At that point, a set of m nodes picked from the previous L (which are in the current set M) are selected as an optimal set of nodes for execution.

To sketch a proof that this algorithm returns the optimal set of m nodes contained in G, assume that there exists a set M' in G such that the lowest bandwidth link in M' has greater bandwidth than the lowest bandwidth link in M. For M' not to have been returned by the algorithm implies that at some point M''s lowest bandwidth link was removed in step 2. However, this could not have happened because the lowest bandwidth link in M would have been removed first, because it has the lower available bandwidth. Therefore, by contradiction, this algorithm cannot return a set other than the optimal set M.

4.3.2 Experimental setup

The node selection procedures presented in this section have been implemented and tested on a networking testbed at Carnegie Mellon. To validate the algorithms, node selection was performed in the presence of realistic computation and communication loads. We describe the setup for experiments and then present results.

Network testbed

All experiments were performed on a part of a networking testbed at Carnegie Mellon. The part of the testbed used for these experiments employs DEC Alpha compute nodes, Cisco routers, and 100 Mbps ethernet links with one 155Mbps ATM link.

Load and traffic generators

Procedures for automatic node selection should do a good job with realistic loads and traffic on the network. But it is virtually impossible to define what is *realistic*, as the load and traffic conditions vary dramatically in network environments. For the purpose of obtaining credible results, we used the results of recent research in characterizing resource usage patterns, and set parameters intuitively to reflect a testbed that is used primarily for data and compute intensive computations.

A synthetic compute intensive job was periodically invoked on every node. Processor load was generated using models developed by Harchol-Balter and Downey, whose measurements indicate Poisson interarrival times, with job duration determined by a combination of exponential and Pareto distributions [54]. Because we are interested in environments which support compute and data intensive computations, higher parameters were used for the load generators than would be used to represent typical interactive systems. Assuming that our target environment is a cluster or group of workstations in a single department, the workload distribution study by Harchol-Balter should be accurate, because their model was derived from observations in such an environment.

For generating network traffic, messages were periodically sent between random nodes. Message interarrival times were Poisson, with message length having a LogNormal distribution. The bulk of the research in network modeling has focused on Internet-level traffic representation, rather than for local area networks. Although there are problems with using Poisson interarrival times for representing bulk traffic and some characteristic of aggregated traffic, it represents the interarrival times of the large high-speed data transfers we would be most concerned about in our target environment rather well [90, 91].

A full validation of the strengths and weaknesses of our techniques would require a large number of experiments with different network usage models and different parameters.

Application		Execution Time with External Load and Traffic (seconds)						Reference
		Randomly selected Nodes			Automatically selected Nodes			Execution
Name of Program	No of Nodes	Processor Load *secs.*	Network Traffic *secs.*	Load+ Traffic *secs.*	Processor Load *secs.*	Network Traffic *secs.*	Load+ Traffic *secs.*	time on Unloaded Testbed
FFT (1K)	4	112.6	80.3	142.6	82.6	64.6	118.5	48
Airshed	5	393.8	281.3	530.2	254.0	188.5	355.1	150
MRI	4	683	591	776	594	571	667	540

Table 4.1: Performance in the presence of computation load and network traffic with automatically selected nodes and random nodes

While we are not at this stage in our experiments, we believe that a study with load and traffic generators that are realistic in some environments does establish the fundamental value of our node selection procedures.

4.3.3 Results

We employed the following 3 applications: 2D fast Fourier transform (32 iterations), Airshed pollution modeling (6 hour simulation) [108], and magnetic resonance imaging(*epi* dataset) [40, 50] to validate our decision procedures. Each application was executed several times with the computation load generator on, network traffic generator on, and with both generators on. Node selection was alternately made randomly and with our automatic node selection procedure. Our experience and previous results indicate that random node selection and node selection based on static network properties give virtually identical performance on a small testbed with all high speed links like ours [74], and hence the random selection results also apply to static node selection procedures. On a small network such as this, benchmarking could be used to determine the location of the bottlenecks. We chose not to examine such techniques because we were interested in techniques that would scale well to larger environments. The results are presented in Table 4.1. Each measurement is the average of a number of executions spanning several hours. Since the activity on the network is changing continuously, a large number of measurements is necessary to have statistically relevant results.

We observe that for all three applications, the load and traffic generators significantly increase the execution time, as compared to the the time with no load (last column of the graph), and their combined effect is cumulative. The impact is fairly high for the FFT and Airshed programs (range of 300% with both generators on and execution on random nodes) but relatively modest for MRI(maximum of around 25%). The reason is that the FFT and Airshed programs are loosely synchronous parallel computations where any computation or communication step can become a bottleneck, while MRI uses a master-slave protocol for compute intensive regions that automatically adapts if a compute or communication step slows down.

In each of these cases, automatic node selection reduces the execution time significantly as compared to random node selection, specifically 8-14% for MRI, 16-23% for FFT and

32-35% for Airshed. We now focus on the increase in execution time due to traffic and load. When using random nodes and with both traffic and load generators on, the FFT time went up from 48 to 142.6 seconds (201%), Airshed from 150 to 530.2 seconds (253%) and MRI from 540 to 776 seconds (43.7%). Correspondingly, with automatic node selection, the increase in execution time was 145% for FFT, 103% for Airshed, and 23% for MRI. Making similar comparisons for other cases, we come to the result that **the increase in execution time due to traffic and/or load is approximately cut in half with automatic node selection**. While we should caution that this result is certainly not applicable to all applications or network conditions, it clearly demonstrates that our node selection procedures are effective in reducing the impact of link and processor sharing on applications.

4.3.4 Related work

Scheduling applications over wide-area distributed systems has attracted considerable attention. Program archetypes are an attempt to develop a framework for building application-class specific parallel adaptive code [27, 31, 76]. More recently, AppleS [11, 12] has been designed for application-centric scheduling of tasks over heterogeneous wide-area networks. It relies on Network Weather Service [117] for resource information. Research with similar goals in the Legion framework is described in [114]. In the process of application scheduling, these systems also address the problem of selecting nodes for execution.

Many application-specific network measurement and adaptation systems have been developed, some examples being [16, 58, 98, 111]. An important goal of this research is to develop a framework that can be used by a large class of applications. A shared memory based approach to adaptive parallelism is explored in [96].

Node assignment and scheduling algorithms in the literature typically do not treat communication in realistic detail, but some recent exceptions are [13, 109]. Several runtime support systems have been developed for partitioning and scheduling computation and communication, an example being [100]. However, the primary job of theses systems is not node selection but application scheduling.

4.3.5 Lessons from cluster selection

Automatic selection of network nodes for parallel and distributed programs is a hard problem and this work introduces a solution framework with a new node selection algorithm. We have made a number of assumptions in order to develop a manageable solution, and certainly more research and experimentation is needed for a more general solution to this problem. However, we have obtained good results on real applications under a realistic load and traffic scenario. This is a tough and realistic way to validate this research, even though it does not establish the generality of the techniques. We specifically demonstrate that our load selection framework was effective in halving the effect of network congestion and machine sharing on application turnaround time. Hence, we believe that we have a good solution and this work represents an important step towards making networked systems like workstation clusters and metacomputers a practical and attractive platform for performance sensitive applications.

The node selection algorithms presented in this section are an excellent demonstration of the usefulness of the topology information available through Remos. Because the network information is available in a graph form, rather than as pairwise bandwidth observations, it is possible to design algorithms that rely on linear operations to determine the minimum bandwidth link, rather than forming crude groups of nodes based on similar performance, such as in ECO. This is a good illustration of a technique that simply can't be accomplished with only higher-level, network as a cloud, measurements.

4.4 Local gap example

The local gap is a parameter designed to represent the communication support distributed systems offer for applications using neighborhood communication. The local gap presents an important juxtaposition between the requirements for low-level network information and dealing with the application at a high level. On the one hand, the local gap cannot be calculated without the detailed topology, capacity, and utilization information about a network that can only be obtained through direct access to the network components themselves. On the other hand, the local gap itself is intended for use as a high-level parameter, describing with only one number the performance of an application on a particular network. The local gap is especially interesting for my thesis because it demonstrates the importance of low-level knowledge even when building high-level support tools.

This section introduces the local gap, an extension that allows the LogP model to be applied to heterogeneous systems. For parallel applications requiring only collective and neighborhood communication, this allows the performance of an application to be evaluated across a variety of machines in the heterogeneous network. I describe the local gap and present initial results obtained in simulations that justify its use as a heuristic for evaluating the performance of parallel applications on heterogeneous irregular networks.

4.4.1 Motivation for the local gap

Unlike well-designed MPP networks, large distributed networks may have disproportionate local bandwidths and aggregate bandwidths for global operations. A tree-structured system with the same speed links at all points has good local performance, but suffers from bottlenecks at the higher levels of the tree. On the other hand, modern network technology offers the ability to construct fat trees using faster network connections or aggregated links. A system with slower links attached to the processors than between switches, a situation common in today's LAN models, may experience bottlenecks only at the leaf links connecting to the processors. These bottlenecks are a major failing of network-based computing, but for applications that require little communication or only the type of communication that a particular network can do well, they do not detract from the usefulness of such systems. What is needed for all systems is a metric that predicts the performance of the type of communication that an specific application requires on a particular network.

The natural way to represent this performance is to establish separate parameters reflecting the ability of a network to perform local and global communication. The performance of global communication is largely dependent on bisection bandwidth, and there is little reason to indicate that this should be different for a distributed network. For the local behavior, however, the critical issue is how well the application's structure maps onto the actual topology of the network.

The success of the gap parameter at predicting the behavior of applications on MPPs makes utilizing the same concept with NOWs and other distributed systems appear promising. The gap cannot be applied directly, however, because of its foundation in representing homogeneous systems, such as MPPs, where nodes and links are identical and there are few bottlenecks.

To apply the gap parameter to distributed systems, a new term called the "local gap" is introduced to represent the gap experienced when performing a computation using neighborhood communication. Local gap is denoted g_i, where i is the degree of tasks in the application structure. Its meaning is identical to the standard gap, except that it considers only nearby processors, whereas the standard gap considers communication with all processors. The standard gap will be denoted g_∞ to retain clarity.

The gap is the ratio between computational speed and communication bandwidth. To determine the local gap of each processor, its computation speed is divided by the bandwidth available per task edge when the task graph is mapped to the network topology.

Because of the inherent asymmetry of networks, the local gap may not be uniform across the entire system. One would generally expect the minimum local gap to be used to represent the entire network, since these processors would be a bottleneck in most computations.

4.4.2 Determining the network load

The first step in calculating the local gap is determining the bandwidth available for each communication edge in the task graph once it is mapped to the network. Once a mapping is determined, the load factor, or number of communication edges sharing a physical network component, can be determined for each network component. The load factor can be used to predict the bandwidth available for the application's communication requirements. I will refer to the set of load factors on all components of the network as the network load.

By its definition, the network load can only be strictly determined when given both the set and topology of processors and a mapping from the structure to be used by an application to that set of processors. Because it is infeasible to do this while making scheduling decisions, it is necessary to provide an approximation without calculating an exact mapping so the network load can be determined quickly. To calculate the network load efficiently, I propose a heuristic method which makes use of the general characteristics of the application's structure.

Both communication requirements and network performance have been divided into local and global components. For network performance, the local aspect is the set of bandwidths at the leaves of the network where the processors are and the global aspect is the set of bandwidths higher in the network, connecting portions of the network. The characteristics of the application's structure will also be divided into these two categories so the calculations naturally match the structure of the network.

An important characteristic for determining the local performance of an application on a network is the degree of the tasks in the application's structure. Task degree may be fixed, as in the case of a torus, which has no external nodes, or it may be variable, as in the case of a mesh without periodic boundaries, which has nodes along its border. The degree of a task in its communication graph determines the load factor placed on the link connecting it to rest the network. For preliminary purposes, a fixed "characteristic" degree will be chosen.

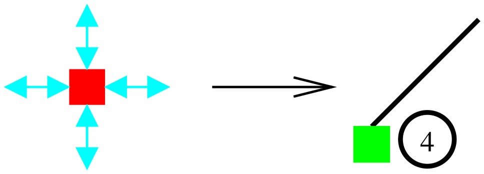
To determine the global characteristics of a structure, consider how it would be partitioned into sets of k and $n - k$ tasks. A straightforward technique for partitioning a mesh, for instance, is to simply divide along an appropriate row or column. Optimal solutions are possible, but a scheduler may not be able to achieve optimality for each partition when scheduling an entire application. It is desirable that the partitioning be something that a reasonable scheduler could actually achieve, rather than being an optimal partitioning, such as partitioning a mesh into rectangles, which may only work under certain conditions, or an extremely bad partitioning with non-contiguous portions [28]). The simple split partitioning works well for small numbers of processors, but as n grows larger, bulk properties begin to dominate, so a more appropriate partitioning technique will be used.

Both local and global structure characteristics can change as the number of tasks in the graph increases, so it is important to consider these effects if the set size is not known in advance, as decisions made when beginning to select a group of processors may have different ramifications as the group expands. It should also be noted that the above discussions assume that the number of tasks in the application structure under consideration are approximately equal to the number of processors. Many applications may be able to distribute their load between processors in a much finer grain.

4.4.3 Heuristic for approximating the network load

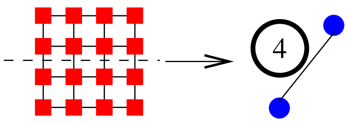
There are three major components to a network topology: processors, switches, and links. Each component can be a bottleneck to the flow of data. Using the characteristics of the structure, it is possible to determine the load factor for each network component so that the total network load can be calculated.

Processors



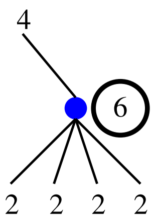
The load factor on a processor is simply the characteristic degree of the tasks in the structure.

Links



The load factor on a network link that partitions the network into k and $n - k$ processors is determined by the partitioning heuristic described above. The load factor will never be lower than the characteristic degree of tasks.

Switches



The load factors across switches are important because some network switches can connect to a higher aggregate bandwidth of links than they can internally support. Also, switches may be used to represent the behavior of shared media. To calculate the load factor on a switch, it is sufficient to find the sum of load factors of all links incident on the switch and divide by two. Since switches can never send or receive messages, this determines the expected load factor for the switch. When switches are attached directly to processors, the average degree of nodes is used rather than the characteristic degree. This is because switches that are attached to several processors are more likely to experience the average processor behavior than to have to deal exclusively with outliers.

4.4.4 Local gap

Once the network load has been calculated, calculating the local gap is simply a matter of calculating the ratio between the computational power and per-channel bandwidth in the network. If the application is synchronous and incapable of load balancing, then the local gap is the minimum computing power of the nodes divided by the bandwidth of the bottleneck link in the network. Using typical metrics, this has units of flop/byte.

In the more common case, where the application is capable of performing load balancing, then the computational work, and, correspondingly, the communication, may be unevenly spread across the network. To calculate the local gap in this case, consider each link in the network. Calculate the average CPU power on each end of the link, and take the maximum of the two values. Then, divide this number by the bandwidth per channel on the link. This gives the local gap for each link. The local gap across the entire network is the maximum local gap for any component of that network.

4.4.5 Simulated verification

To verify the usefulness of the local gap, I have run a series of simulations using randomly generated network graphs. The use of simulations allows the heuristic to be compared across widely varying network topologies—much more varied than would be easily possible on real networks.

Topology generation

To generate the network topologies used for the simulation, first ten graphs were generated for each size network. Each graph was annotated with six different bandwidth assignments. This combination resulted in sixty total network topologies providing a variety of performance characteristics.

The graph generation algorithm was designed to produce a variety of realistic tree topologies. As such, the algorithm was designed to produce a network where CPUs are connected only to switches on the leaves of the topology. These leaf switches are then connected with internal switches that only connect to leaf switches—there are no internal CPUs.

Specifically, the algorithm first generates the leaf switches with a normally distributed number of CPUs attached. To achieve the exact number required in the network, if it generates too many CPUs, it selects a previously generated switch to discard. This process is continued until it achieves exactly the target number of CPUs.

To connect the leaf switches, the algorithm first places all previously generated switches in an *UnconnectedSet*. It then creates a new switch and selects its degree as a uniformly distributed number between 2 and a specified maximum switch degree. This number of switches are randomly removed from *UnconnectedSet* and connected to the new switch. If there are not enough switches in *UnconnectedSet*, then all remaining members are used. The new switch is then added to *UnconnectedSet*. This process is repeated until there is only one switch in *UnconnectedSet*, which will be designated the root of the graph.

For these experiments, leaf switches had $N(5, 3)$ CPUs attached to them. Internal switches connected between 2 and 4 switches.

The links in each graph were then assigned six different bandwidth combinations. For these experiments, the links are bidirectional, with the same bandwidth in each direction. To simplify the scheduling problem, all CPUs attached to a leaf node have the same bandwidth. If it is changed, that bandwidth is changed for all nodes. The combinations were:

Fixed Each link in the graph is assigned the same bandwidth, here $U(1, 25)$MBs.

Fixed with outliers The same graph as before, but with two randomly selected links given bandwidths a factor of $U(2, 4)$ higher. To select a link, first a switch is randomly selected. If that switch is a leaf switch, then the bandwidths of the links connecting to its CPUs are raised. If the selected switch is an internal switch, then one of the links connecting it to another switch is selected.

Fixed with bottleneck The first graph again, but with one randomly selected link having its bandwidth lower by $1/U(2, 4)$.

Random Each link in the graph is assigned a bandwidth of $U(2, max)$ where $max = U(1, 25)$MBs for all links.

Proportional Each link the graph is assigned a bandwidth proportional to the minimum number of CPUs found on either end of the length. The base bandwidth here is the bandwidth of the fixed graph divided by three.

Proportional with bottleneck The previous graph with one randomly selected link having its bandwidth lower by $1/U(2, 4)$.

Application

The application used for this experiment is FDTD, an electromagnetic field solver used to study antenna patterns, calculate electromagnetic scattering from targets, and examine fields within small circuits and boards [63, 64]. The code uses a uniform three dimensional

Cartesian grid with staggered cells and is divided among processors on a two dimensional grid. Each processor maintains ghost cells at its boundaries to minimize the effects of network latency. The code was originally developed for use in a Beowulf cluster. It was one of the early programs developed for Beowulfs that showed equivalent performance to a T3D MPP at substantially lower cost [65].

Because only the communication was of interest for these simulations, the application essentially consists of tasks communicating along a non-periodic mesh, with each message being 228528 bytes long. When run on a network of 300 Mhz Alphas connected via 100Mb switched Ethernet, the problem size generating this data executes at approximately 85% efficiency. Note that because the message size was fixed as the number of processors increased, this is equivalent to increasing the problem size proportionately to the number of processors being used.

Application mapping

As mentioned previously, the application mapping algorithm requires the bandwidths of each link connecting a CPU to its leaf switch to be the same for all CPUs connected to that leaf switch. This allows the mapping algorithm to simply assign processes to the leaf switch, without having to further optimize the mapping. The mapping algorithm relies on a greedy heuristic to do the initial task assignment, followed by swapping to improve performance.

The initial assignment is done by sorting the tasks from highest to lowest bandwidth requirement, selecting randomly in the case of ties [62]. For this application, the internal tasks have identical bandwidth requirements, followed by boundary tasks. After selecting a task, it first checks if any messages are exchanged with tasks already assigned. If so, it assigns the new task to the leaf switch with CPUs for additional tasks and with the tasks with which the most data is exchanged. If an assignment cannot be made with this rule, then it assigns the task to the leaf switch with the highest bandwidth connection, preferring leaf switches without any tasks already assigned.

After the initial assignment, a deterministic swapping algorithm is performed to reduce the bottleneck link in the resulting network load. First, the most congested link in the network is found. The tasks exchanging messages across that link are identified. The algorithm examines each pair of tasks, each pair having one task from each side of the link, and determines which swap most reduces the traffic on the bottleneck link. This swap is then made to the mapping. This algorithm is repeated until it can make no further improvements to the bottleneck link on the graph.

The final phase of the algorithm implements a randomized swapping algorithm similar to that described by Kernighan and Lin [66]. The algorithm first marks all tasks as unswapped. It then randomly selects an unswapped task. Next, it tries to swap that task with all other unswapped tasks. It makes the swap that has the best performance and removes both swapped tasks from the unswapped list. This is repeated until all tasks have been swapped. Note that this algorithm may select to make a swap that actually makes the performance of the task assignment worse. This behavior is an attempt to avoid local minima in the mapping's performance.

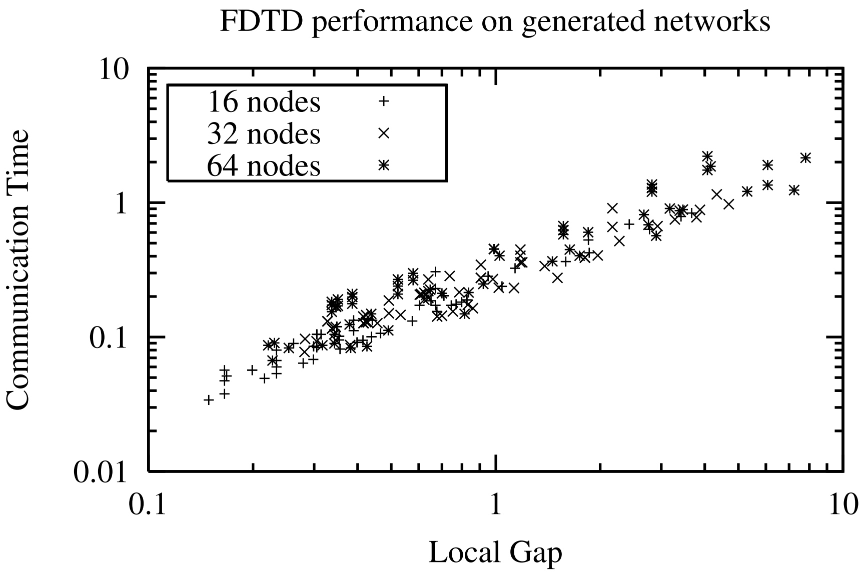


Figure 4.7: Communication time versus local gap for the FDTD application on randomly generated networks of 16, 30, and 64 nodes.

Although a number of the ideas used in these three phases of the mapping algorithm are based on other published mapping algorithms [28, 62, 66], this is not a reimplementation of any specific published algorithm. It is certainly not guaranteed to provide optimal results. In bench checking this implementation on several randomly generated 16 and 25 node mappings, this implementation always performed as well, or better, than my attempt to determine the optimal mapping.

For the experiments described here, the entire mapping procedure was run several times for each generated network, to determine as close to the optimal mapping as possible.

4.4.6 Simulation results

Figure 4.7 presents the combined results from the simulation experiments. For these results, the CPU power was fixed at 1, representing uniform computing power on every node. A more thorough evaluation would require this variable to be relaxed. In this case, because the application, as simulated, did not support load balancing, the only factors of interest in performing the mapping are the minimum CPU power and the bottleneck bandwidth, which are used to determine the local gap.

The linear fit of the results, and especially its consistency across three different application sizes, is an extremely promising result for the local gap. Further work, and corroboration with real experimental work, is needed to validate its use as a significant metric of the capabilities of distributed systems.

4.4.7 Lessons from the local gap

The local gap is an excellent example of how the detailed topology information made available through Remos can be used to meet an application's high-level needs. In many cases, an application programmer may not want to manage the level of detail offered through the topology interface. Even in these cases, however, the information about the topology can be of use. The local gap is a very simple metric that applications can use to make decisions, but it reflects details that are simply impossible to calculate without topology knowledge.

4.5 Ethernet topology discovery algorithm

As described earlier, the challenging problem in topology discovery is dealing with Ethernet LANs. However, the previous examples all require LAN topology information, so it is necessary to develop an algorithm that discovers Ethernet topology in spite of the transparency designed in the Ethernet protocol.

A bridged network containing the nodes N can be divided into a set of bridges, B, and endpoints, E. E consists of both hosts and routers, which are identical to hosts for the purpose of the Ethernet bridging algorithm. Bridged Ethernet networks may also include hubs, which serve to connect several machines off of one port of a bridge via shared Ethernet. Hubs merely join a segment of shared Ethernet and are logically just another address on that shared segment. Although hubs are physically on the internal portion of the Ethernet tree, they are actually considered endpoints because they act as just another machine on that shared segment.

For each bridge, A, consider the set of addresses sending messages received on port x as F_A^x. This is the set learned by the transparent bridging algorithm described above. This set changes as new addresses are learned, as old addresses expire, or as machines are physically moved to part of the network connected to a different port. F_A^x is said to be *complete* if it contains a forwarding entry for each member of N found off of that port.

I will first describe the intuitively obvious way to use this information to determine the topology of the network. The basis of this algorithm is determining which nodes are directly connected. Two nodes are referred to as *directly connected* if there are no other nodes between them. In particular, if packets bridge A sends on port x are received by port y on bridge B without going through any other device, bridges A and B are directly connected via ports x and y. Those two bridges are *connected* by ports x and y if they find each other with those ports, but there may be other nodes in between.

At a high-level, this algorithm begins by determining all entries in the FDBs, ensuring they are complete. It then selects a single bridge and determines the bridges that are directly connected to each port. The Ethernet topology is then filled in by traversal. At the heart of this code is the direct connection theorem, which is used to determine when two bridges are directly connected.

Theorem 4.1 (Direct Connection Theorem) *Assume that F_i^x and F_j^y are complete. Two bridges i and j are directly connected via the link connected to port x on i and port y on j if and only if $F_i^x \cap F_j^y = \emptyset$ and $F_i^x \cup F_j^y = N$.*

Proof $\Leftarrow$: Assume that $F_i^x \cap F_j^y = \emptyset$ and $F_i^x \cup F_j^y = N$. For that to be true, F_i^x and F_j^y must be a partitioning of N. The set N must contain at least i and j, which rejects the trivial solution. Because a bridge cannot hold its own forwarding entry, for $F_i^x \cup F_j^y = N$ to hold, F_i^x must contain an entry for j and F_j^y must contain an entry for i. Therefore, the bridges are connected through the links attached to ports x and y. For $F_i^x \cap F_j^y = \emptyset$ to hold, there can be no other nodes in between i_x and j_y, because if there were nodes in between they would be in the intersection because F_i^x and F_j^y are complete and would both have entries for reaching the nodes in between. Therefore, i and j are directly connected if $F_i^x \cap F_j^y = \emptyset$ and $F_i^x \cup F_j^y = N$.

$\Rightarrow$: First assume that i and j are directly connected. Recall that bridged Ethernet topology is defined to form a tree. Partition the tree across the link directly connecting the two bridges. Denote the partition containing i as N_i and the partition containing j as N_j. Due to the acyclic requirement of the Ethernet topology and the completeness requirement of the theorem, $F_i^x = N_j$ and $F_j^y = N_i$. Because N_i and N_j are a partitioning of N, $N_i \cap N_j = \emptyset$ and $N_i \cup N_j = N$. Therefore, $F_i^x \cap F_j^y = \emptyset$ and $F_i^x \cup F_j^y = N$. This proves that $F_i^x \cap F_j^y = \emptyset$ and $F_i^x \cup F_j^y = N$ if i and j are directly connected. ■

For a simple switched Ethernet, Theorem 4.1 is sufficient, but for many real networks, there are cases that are not addressed by it. One common cause of failure is a shared segment internal to the bridged topology. Shared segments can occur in two situations:

1. A hub is used to connect two bridges with other hosts or bridges. Because hubs do not participate in the bridging algorithm, this creates a shared network segment between the bridges. The reality is that a properly designed network would never contain such a hub, but the real world does not guarantee that all networks will be properly designed.

2. A bridge exists in the network that is not a member of B. This situation can occur either when the program is not informed of the bridge's existence or when SNMP access is denied to that bridge. Because SNMP protection generally consists of a simple allowed/denied list, this situation can easily occur and did, in fact, occur on a number of occasions during the development of the topology discovery system.

Because Theorem 4.1 detects only direct connections and not bridges connected with shared segments, a new rule is required. To develop this rule, first let $a(b)$ be the port of bridge a that address b is found off of. Let S_B be the set of bridges connected to the shared segment. (Note that $|S_B| > 1$.) Let S_E be the endpoints attached to the shared segment. The entire shared segment is denoted $S = S_B \cup S_E$.

Theorem 4.2 (Shared Segment Theorem) *S consists of a shared segment between the bridges in S_B if and only if $\forall a \in S_B, \forall b, c \in S : a(b) = a(c)$ and all forwarding databases are complete.*

In other words, on a shared segment, all bridges must find all members of the shared segment on the same port.

Proof $\Leftarrow$: Assume that $\forall a \in S_B, \forall b, c \in S : a(b) = a(c)$, but S_B are not connected with a shared segment. If S is not shared, then there exists a switch in S_B that has elements of S on different ports, by the definition of a shared segment. More formally, and by applying the completeness requirement, $\exists a \in S_B, \exists b, c \in S : a(b) \neq a(c)$. This contradicts the original assumption. Therefore the bridges are connected with a shared segment if $\forall a \in S_B, \forall b, c \in S : a(b) = a(c)$.

$\Rightarrow$: Assume that the members of S_B are connected via a shared segment containing S_E. By definition of a shared segment, no bridges in S_B may have elements of S on two different ports, otherwise, by the acyclicity requirement, the segment would be switched, rather than shared. Therefore, every member of S_B has all members of S on the same port. Stated formally, $\forall a \in S_B, \forall b, c \in S : a(b) = a(c)$ if the bridges are connected with a shared segment. ■

4.5.1 Implementation

The first step of implementing this algorithm is acquiring complete FDBs from all bridges involved in the topology. This algorithm begins with a set of endpoints, E, consisting of all of the hosts on the Ethernet for which the topology is desired, as well as the routers used to connect this Ethernet to other networks. The set of bridges used in this network, B, must be known. The basic approach to determining this topology is to go through the members of B, querying for the ports to which they forward packets routed to members of $E \cup B$.

Because bridges learn passively, each bridge in B must have seen a packet from each member of $E \cup B$ to have an entry in its forwarding database for that node. To ensure that this table is complete, all members of E periodically ping all other members of E and B before and during the data collection. This guarantees that if a bridge is on the topology between any two members of E, it will have seen packets from both members and will have their entries in its forwarding database. Note that it is generally not possible for users to have routers or bridges send pings, but routers do respond to pings, so if all hosts are sending pings to a router, the router's replies to the pings will ensure that each bridge's FDB has an entry for that router.

If a bridge is not used in forming the topology between the members of E, then all entries in its FDB for members of E will point to the same port. The algorithm drops these bridges from the network's topology because they are unused in the part of the network that is of interest in the query.

After the complete FDBs are acquired, the algorithm calculates the topology. It begins with a single bridge and applies Theorem 4.1 to determine which other bridge is directly connected to each port. If it finds that at least one bridge is found off of a port, but no bridge is directly connected, then it applies Theorem 4.2 to determine which bridges are connected to the shared segment.

The topology graph is traversed, applying this algorithm to each bridge in the topology, until all connections are resolved.

4.5.2 Failures of the direct connection theorem

While this approach works, there are several shortcomings. The most significant is the requirement that all members of E be running programs to send pings. This requirement restricts the topology discovery to environments where all nodes support remote login and where the user has an account on each machine. If several members of E are routers or run Microsoft Windows, this requirement cannot easily be met. Even in other environments, rarely does a single person have accounts on all machines. While an individual could use this algorithm to determine the topology connecting their own machines, a shared database or a database for machines that are not always available would be impossible. Even if it is possible, the load and overhead of 1000 machines sending regular pings between each other would be significant.

A second problem is that this algorithm requires every bridge's forwarding database to be complete. While obtaining complete FDBs may be possible when sending this many pings, various practical difficulties with bridges make it difficult to obtain such information reliably.

The most significant difficulty with obtaining complete FDBs is that some bridges are connected via out-of-band ports. In other words, they communicate with other machines in the network through an interface not involved in forwarding packets. While out-of-band connections allow the network manager to contact the bridge even when the rest of the network is not functional, they also allow packets sent to the bridge to follow a different path to the bridge than one on which packets destined for endpoints would find the bridge. If a bridge is connected using an out-of-band port, the information provided for that address cannot be used to determine where the bridge is placed in the topology graph. This prevents the completeness requirements of the direct connection theorem and shared segment theorem from being met.

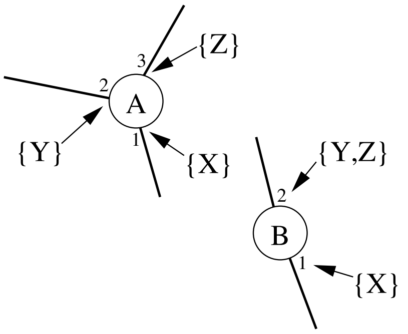
More commonly, it is tremendously difficult to obtain complete sets for many bridges. The code used to implement this algorithm required several attempts to obtain complete sets, even for relatively small numbers of nodes. The fundamental problem of this approach is that it is not universally possible to obtain complete information about the network.

As the network grows larger, the problem of obtaining complete forwarding sets becomes even more challenging. In a large network with hundreds or thousands of machines, more and more of the machines will be down or unresponsive at a given time, whether do to hardware failure, software failure, being rebooted, or other problems associated with maintaining a large network. Capturing a complete, consistent picture in a real-world dynamic network environment is less and less likely in this type of dynamic environment.

A workaround to this problem is used by Breitbart et al., who relax the completeness requirement to $|F_i^x \cup F_j^y| \geq (1-\epsilon)|N|$ [20]. While the relaxed rule increases the chance of success, it still requires a tremendous amount of work to generate forwarding entries for most machines in all of the bridges in a large Ethernet.

4.5.3 Related work

The algorithm based on the direct connection theorem was developed by myself for Remos. A sketch of it was published in HPDC8 [75]. My description focused on the discovery of complete FDB entries for the bridges and largely ignored the details of the theorems that allowed the implementation. Independently, Breitbart et al., from Lucent Technologies, developed and published a description of an almost identical algorithm [20]. The princi-



Bridge	Port	Forwarding Entries
A	1	X
	2	Y
	3	Z
B	1	X
	2	Y, Z

Figure 4.8: Example of two bridges for which contradictions can be used to determine the connections.

ples utilized in their algorithm are essentially identical to those of mine, which I had not published. Their formal description of the algorithm, however, was much more complete than what I had written in the development of my algorithm.

There are important differences between our techniques, however. The researchers from Lucent focused on Ethernet networks with multiple IP subnets and VLANs. They found that by extending the algorithm with information available from non-standard vendor MIBs, their algorithm could handle topology discovery on networks using VLANs.

On the other hand, their algorithms do not address the problems of shared segments appearing between bridges, whereas I encountered a number of these shared segments and expanded my algorithm to support automatic discovery of shared segments. There were also a number of practical challenges that my code had to deal with to support the Ethernet LANs at CMU that were apparently not a problem for the networks at Lucent. These variances generally seem to correspond to the nonstandard aspects of the networks we each sought to support.

Although there are important implementation and support differences between our different algorithms, the fundamental techniques used by the algorithms remain the same.

4.6 Topology discovery with incomplete knowledge

Because of the difficulties associated with obtaining complete forwarding databases for entire Ethernet LANs, I decided to pursue an alternative approach to topology discovery. Rather than proceeding with the goal to prove that two nodes are directly connected, consider the approach of proving that bridge a is not connected to b_x. Because this is a proof by contradiction, only a single counterexample is needed to demonstrate the contradiction.

Figure 4.8 shows an example where contradictions can be used to determine how two bridges are connected. The motivating concept here is to pretend there is a connection between two specific ports of two bridges. Here I am referring to a simple connection, with other nodes possibly in between, not a direct connection. If all entries in the forwarding database are consistent with this connection, it can be made. If any one entry is inconsistent, then that connection is impossible.

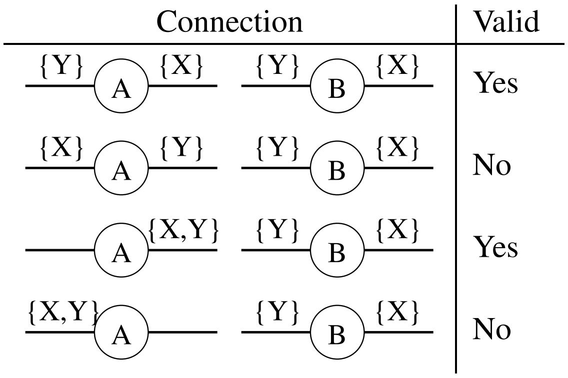


Figure 4.9: Examples of valid and invalid connections between two bridges.

Figure 4.9 presents four examples of valid and invalid connections. The only situation where a contradiction can be found in making a connection is where the two bridges both have a forwarding entry for the same address on ports other than the connecting ports. In other words, they claim the same machine is in two different places on the network. In Figure 4.9 this rule excludes the second and fourth examples, where the two bridges forward the same address in opposite directions. Anytime there is not directly conflicting forwarding entries, the connection may be valid. Note that in the third example of Figure 4.9, node Y is between bridges A and B. Because this rule establishes only a connection and not a direct connection, this is perfectly valid.

Figure 4.10 shows how contradictions can be used to determine the only valid connection between the bridges in Figure 4.8. All six possible connections between the bridges are shown. In five of the six connections, a contradiction of the two bridges forwarding the same address in opposite directions was found. Only the valid connection is left.

The most important observation about the application of this simple rule is that there is no complete knowledge requirement. Instead of requiring complete knowledge, there is merely the minimal requirement of having enough information to rule out all but one possible connection. I will describe and prove what this minimum knowledge requirement is. In the example in Figure 4.10, the entries for these three addresses meet the minimum requirement. There may be many other nodes in the network, and there may be entries for many other nodes on one or both of bridges A and B, but as long as the minimum information is captured as it is in this example, the correct connection will be determined.

To simplify the presentation of the algorithm, I will first introduce additional notation. Consider the network shown in Figure 4.11. In this graph, the bridges in the network are shown annotated with the addresses they learn through the forwarding algorithm, the sets F_i^x of addresses forwarded by each port. Figure 4.12 shows the addresses that are forwarded through each port, in other words, the addresses that are on other ports, for which the bridge will forward packets through itself. The addresses forwarded through each port are, in a sense, the complement of F_i^x and are denoted T_i^x.

After determining the set of addresses forwarded through each port, shown in Figure 4.12, it is simple to apply this information to determine the ports that connect two bridges. To see if ports x and y of bridges a and b are connected, determine T_a^x and T_b^y.

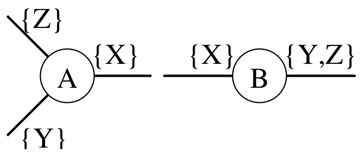
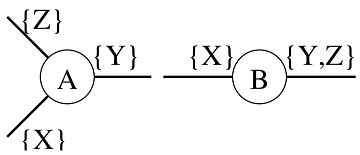
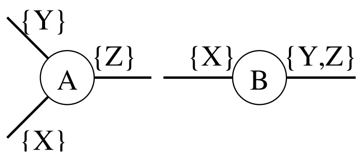
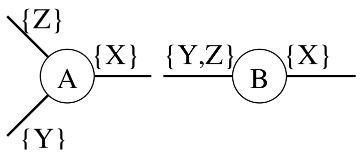
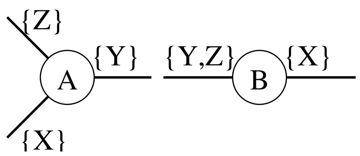
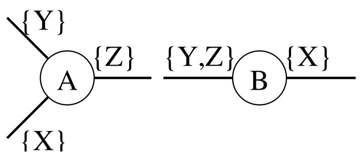
Ports A	Ports B	Mapping	Conflict
1	1		Y and Z
2	1		Z
3	1		Y
1	2		**NONE**
2	2		X
3	2		X

Figure 4.10: How contradictions can be used to eliminate impossible connections from the bridges in Figure 4.8.

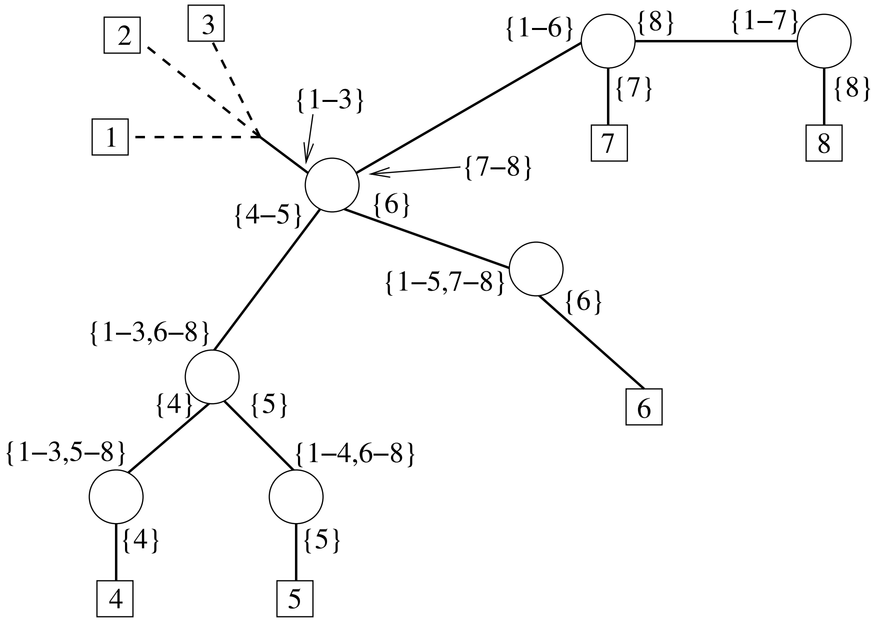


Figure 4.11: The topology of a portion of a bridged Ethernet. The ports on the bridges are annotated with the addressed learned in their forwarding databases, denoted F_i^x. This figure illustrates complete forwarding databases.

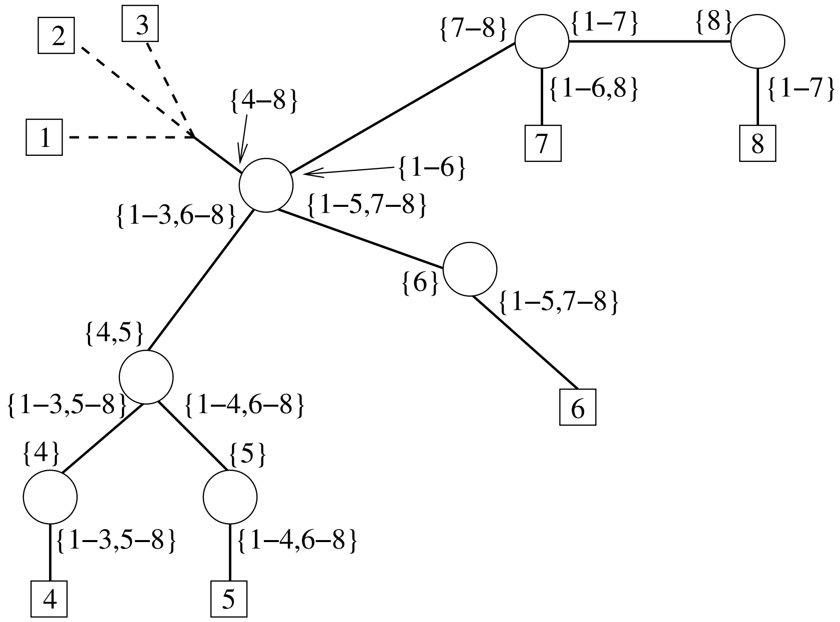


Figure 4.12: The same network as Figure 4.11. In this figure, the ports on each bridge are annotated with the addresses forwarded through that port, denoted T_i^x.

The intersection of these through sets, $T_a^x \cap T_b^y$ represents the addresses that the two bridges forward in different directions. If there are any addresses in common, then the ports cannot be connected because a single node cannot be in two different directions on an Ethernet, which is required to be acyclic. If the intersection is null, they are connected.

By simple iteration, it is possible to map each bridge to the port it appears off of every other bridge. Applying this technique to each bridge allows the complete topology of the network to be determined.

This approach has several advantages to the previous approach:

- Rather than relying only on information about one port, information from all ports is combined for each mapping question. This is especially helpful for ports with few machines connected to them, because it allows data to be aggregated without requiring that it be complete.
- Incomplete information is tolerated much more easily.
- Shared segments are naturally determined from the primary theorem, rather than relying on a special case to resolve conflicts when the direct connection theorem fails.
- The theorem offers positive rejection: if insufficient information is available to perform a mapping, that error is detectable and distinguishable from a case where a mapping is performed with incomplete, although accurate, data.

4.6.1 Rigorous presentation

Theorem 4.3 (Simple Connection Theorem) *Let $a, b \in B$. Suppose there exists exactly one pair of ports a_x and b_y such that $T_a^x \cap T_b^y = \emptyset$. Then a_x and b_y are connected. Furthermore, if a_x and b_y are connected, then $T_a^x \cap T_b^y = \emptyset$.*

Remark: Note that a_x and b_y being connected does not imply that $T_a^x \cap T_b^y = \emptyset$ for only one pair x and y. See Figure 4.13 for an example.

Proof $\Rightarrow$: Assume that a_x and b_y are connected. Partition the network into three partitions, let N_S be the partition between a_x and b_y, let N_a be the partition containing a, and let N_b be the partition containing b. $T_a^x \subseteq N_a$ and $T_b^y \subseteq N_b$. Because N_a and N_b are a partitioning of N, $T_a^x \cap T_b^y = \emptyset$. Therefore, $T_a^x \cap T_b^y = \emptyset$ if a_x and b_y are connected.

$\Leftarrow$: Assume that $T_a^x \cap T_b^y = \emptyset$ for only one pair x and y, but a_x and b_y are not connected. Because a and b belong to a connected Ethernet, they must be connected by some pair of ports. Let a_i and b_j be the true ports connecting the bridges. From the first half of the proof, $T_a^i \cap T_b^j = \emptyset$. However, this contradicts the assumption that the intersection is null for only one pair of ports. Therefore, by contradiction, a_x and b_y are connected if $T_a^x \cap T_b^y = \emptyset$ for only one pair x and y. ■

The final portion of Theorem 4.3, that the intersection is null for only one pair x and y, is the minimum knowledge requirement. It serves to prevent trivial solutions to $T_a^x \cap T_b^y = \emptyset$ that exist only because the forwarding databases have insufficient information or because the network topology is structurally indeterminate. Consider the network shown in

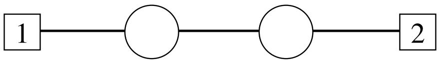


Figure 4.13: In this network, the order of the two bridges cannot be determined because there is no information available using only two hosts.

Figure 4.13. These two bridges can be arranged in either order because the knowledge they give is insufficient to determine ordering between themselves using only the forwarding entries for the hosts shown.

The minimum knowledge requirement can be met by any pair of bridges that meet the following rule:

Lemma 4.4 (Minimum Knowledge Requirement) *The ports x and y that connect a and b are uniquely determined if and only if any one of these conditions is met:*

1. *Each bridge has an entry for the other's address in its FDB; or*

2. *Bridge a has an entry for b in F_a^x and $\exists k \neq x : F_b^y \cap F_a^k \neq \emptyset$; or*

3. $\exists i, j, i \neq j : (F_a^x \cap F_b^i \neq \emptyset \wedge F_a^x \cap F_b^j \neq \emptyset)$*, and* $\exists k \neq x : F_b^y \cap F_a^k \neq \emptyset$.

Proof The first condition is trivial—if each bridge has an entry for the other in their FDBs, then the entries directly indicate which ports are used to connect the bridges. If that option is not available, one of the other two must be met.

For the second condition, port x is uniquely determined to be the port of a connected to b because of the explicit FDB entry. Port y on b is uniquely determined because there is an entry shared between b_y and a port on a other than a_x. If a connection is tried using any port other than y, these two entries will cause a contradiction by pointing in different directions for the same entry. This is the important condition required by $\exists k \neq x : F_b^y \cap F_a^k \neq \emptyset$.

Proving the second half of rule two formally: $\Leftarrow$: Assume $\exists k \neq x : F_b^y \cap F_a^k \neq \emptyset$. Let $c \in F_b^y \cap F_a^k$. Because $k \neq x$, $c \in T_a^x$. Also, $\forall l \neq y : c \in T_b^l$. Because c is in both sets, $T_b^l \cap T_a^x \neq \emptyset$. Therefore, if $\exists k \neq x : F_b^y \cap F_a^k \neq \emptyset$, y is uniquely defined.

$\Rightarrow$: Assume y is uniquely defined, but $\forall k \neq x : F_b^y \cap F_a^k = \emptyset$. Because x and y are connected, $T_b^y \cap T_a^x = \emptyset$. By definition of through sets, $\forall l \neq y, \forall k \neq x : F_b^l \cap F_a^k = \emptyset$. Combined with the initial assumption, $\forall l, \forall k \neq x : F_b^l \cap F_a^k = \emptyset$. Again by the definition of through sets, $\forall l : T_b^l \cap T_a^x = \emptyset$. This contradicts the assumption that y is uniquely defined. Therefore if y is uniquely defined, $\exists k \neq x : F_b^y \cap F_a^k \neq \emptyset$.

The third, and fully general condition is an extension of the logic used for condition two to using intersections to make the unique determination of both ports x and y. The third case requires that F_a^x must have members also found in two ports of b, denoted F_b^i and F_b^j. This condition is sufficient to uniquely determine x. To see why an entry must be shared with two ports, whereas for condition two, a shared entry with only one port was required, consider the case where there is an entry shared between port a_x and port b_i, but no entries shared with any port other than b_i. In this case, a_x can be connected to any port on b without creating a conflict. However, b_i can also be connected to any port on a without creating a conflict, as the shared entry will simply be indicated to forward through

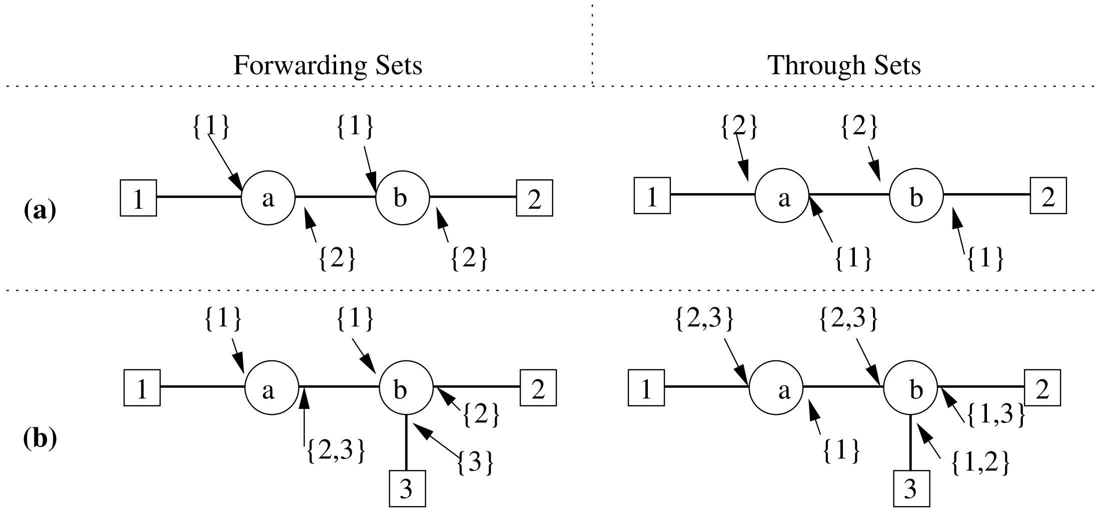


Figure 4.14: Two network graphs are shown with forwarding and through sets indicated. Network (a), as seen in Figure 4.13 does not provide sufficient information to meet the minimum knowledge requirement. Network (b), however, with the addition of one additional host, meets that requirement.

a to port a_x. So this single shared entry does not uniquely define x. Now consider the case required in the third condition, where there is also a shared entry with b_j. In this case, if a_x is connected to b, the connection is valid. However, if any port of a other than a_x is used for the connection, a conflict is created. If b_i is connected to a port on a other than a_x, a conflict will be created with the entry shared with b_j, because they will be forwarding the same address in different directions. Likewise, a conflict will be created if b_j is connected to a port other than a_x. If a port on b other than b_i or b_j is connected to a port other than a_x, then the entries from both b_i and b_j will create a conflict. Therefore, we see that these two shared entries uniquely determine a_x as the port used to connect a to b.

Once x is uniquely determined, y is determined with the same rule as used for condition two.

As a further example, consider the graphical illustration of an indeterminate network shown in Figure 4.14. In network (a), there is not a unique solution to the mapping problem. The bridges share two entries, but they are symmetric and can be placed in either order. Network (b), however, resolves the indeterminism. By adding a second machine off of one of the bridges which is shared in both bridges' FDBs, the network is uniquely determined. This second machine gives a shared entries with two of b's ports. It is no longer possible to reorder these two bridges because the only port on the left bridge that can be connected to the right one is the one with both entries. Any other connection will cause a contradiction.

The formal proof of condition three begins by proving that $\exists i, j, i \neq j : (F_a^x \cap F_b^i \neq \emptyset \wedge F_a^x \cap F_b^j \neq \emptyset)$ uniquely determines x.

$\Leftarrow$: Assume $\exists i, j, i \neq j : (F_a^x \cap F_b^i \neq \emptyset \wedge F_a^x \cap F_b^j \neq \emptyset)$. This implies that $\forall z$: $F_a^x \cap T_b^z \neq \emptyset$ because either F_b^i or F_b^j will be in T_b^z. Accordingly, $\forall l \neq x, \forall z : T_a^l \cap T_b^z \neq \emptyset$. Therefore, x is uniquely determined if $F_a^x \cap F_b^i \neq \emptyset$ and $F_a^x \cap F_b^j \neq \emptyset$.

$\Rightarrow$: Assume that x is uniquely determined, but that $\forall i \neq j : F_a^x \cap F_b^i = \emptyset \vee F_a^x \cap F_b^j = \emptyset$. In other words, there is at most one i such that $F_a^x \cap F_b^i \neq \emptyset$. Consider two cases:

- First, that there is no set F_b^i such that $F_a^x \cap F_b^i \neq \emptyset$. We know that $T_a^x \cap T_b^y = \emptyset$. Combining these two, $\forall l, \forall i : F_a^l \cap F_b^i = \emptyset$. Therefore, there are no common members, which contradicts the initial assumption that x is uniquely determined.

- The second case is that there exists only one i such that $F_a^x \cap F_b^i \neq \emptyset$. Note that network (a) in Figure 4.14 meets this condition, but is indeterminate. By example, this is a contradiction.

Therefore, if x is uniquely determined, $\exists i, j, i \neq j : (F_a^x \cap F_b^i \neq \emptyset \wedge F_a^x \cap F_b^j \neq \emptyset)$.

Once x is uniquely determined, the proof of rule 2 proves y is uniquely determined if and only if $\exists k \neq x : F_b^y \cap F_a^k \neq \emptyset$. ■

Therefore, it has been shown that Lemma 4.4 is truly the minimum requirement for Theorem 4.3 to be able to determine the connection between two bridges in a network.

4.6.2 Practicality

The next question to be asked is whether Lemma 4.4 is a realistic expectation for bridged Ethernet networks to meet. After all, the motivation for pursuing this technique is that it is much easier to satisfy the minimum knowledge requirement than to require the FDBs to be complete.

Consider the four snapshots of two bridges shown in Figure 4.15. Imagine that the real position of these two bridges has bridge A internal to the topology and bridge B positioned as a leaf bridge, connecting only to one other bridge with the remainder of its ports connected to endpoints.

In Figure 4.15(a), only one host off of the lower bridge is shared. This is obviously indeterminate. In Figure 4.15(b) the new address 2 might correspond to the querying node sending pings to address 1 while probing the FDBs. However, for the same reasons as Figure 4.13, this mapping is still indeterminate.

Now, suppose that node 1 has communicated with some other host on the network, for instance a nameserver. If that node is also found on any port on the upper bridge other than the port with 2 in its forwarding set, such as in Figure 4.15(c), then the mapping is determinate. In many cases, it is possible to force a machine to contact a nameserver by connecting to its FTP daemon, for instance.

In most cases, networks aren't designed with only one endpoint connected to a bridge. There is no reason to purchase a bridge in that case. Almost all bridges have 4, 16, 24, or even more ports used for connections to machines. A bridge with just two ports used will satisfy Lemma 4.4. Consider Figure 4.15(d). The only entries in the FDBs are for endpoints on two different ports of bridge B and for the querying machine sending them pings. Networks (c) and (d) are two minimal examples—a single machine on a bridge that

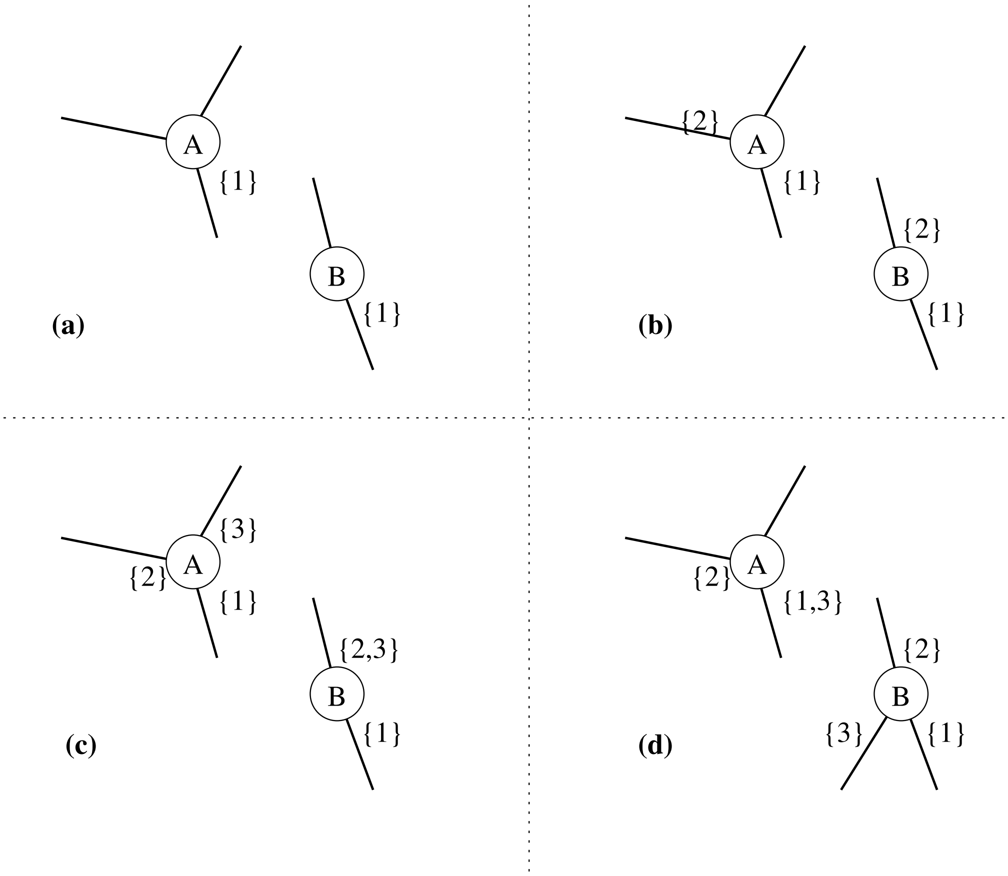


Figure 4.15: These snapshots of two bridges within a network demonstrate different examples of indeterminate and determinate FDBs. The forwarding sets are shown for each port. (a) and (b) give examples of indeterminate networks. (c) and (d) are both determinate, although for different reasons.

has talked to more than one other machine (host, server, or router) or a bridge connected to two machines. Practically every part of an Ethernet will meet one of these criteria. However, because bridges are generally only installed in a network to be used, which means there are machines divided among several ports, most components will provide information well beyond the minimal requirements.

4.6.3 Specialization for traversal

An important specialization of Lemma 4.4 can be used in most cases. If the network's topology is determined by traversal from a designated root, then the first mapping step resolves the ports connecting each bridge to the root bridge. From that point in the algorithm, the "root port" of each bridge is fixed. This knowledge can be exploited by the algorithm. In fact, being able to fix the root port of a bridge satisfies the first half of rule 3 in Lemma 4.4. This reduces the requirement for subsequent mappings to only one entry shared between the connecting port and another port on the child bridge.

This specialization is easy to exploit. The root is picked to maximize the chances that it meets the minimum knowledge requirement. The only complexity in the traversal is that each bridge must be tested to see if it is the next hop in the traversal. Determining which bridge is the next hop is easy, because performing the mapping with an incorrect bridge will either indicate that there are bridges between itself and the root bridge (these bridges will be connected to the incorrect choice's root port), or it will actually cause a conflict because a bridge will have to be connected to it on a port other than its root port, indicating that this isn't the true topology.

In summary, the minimum knowledge requirement is very easy to satisfy. Any bridge that is installed and used to connect more than one machine will meet the requirement. Furthermore, even if the topology has little information, once the root port has been determined, only one entry is needed to meet the requirement.

4.7 Implementation

The actual implementation differs somewhat from the described theorems only in that it tries to provide solutions in situations where there is not enough information to solve the mapping problem deterministically and that it handles mapping endpoints to the bridge topology.

4.7.1 Preparation

The algorithm begins with a set of bridges, B, and a set of endpoints, E. The bridges and endpoints must all be within the same bridged Ethernet. In many networks, this can be determined by IP address and netmask. In proxy-routed networks, such as that used at CMU, only traceroute can determine whether an address is on the local bridged Ethernet or across a router. Some networks may actually involve multiple IP subnets on the same bridged network. This algorithm does not make use of IP addresses at any point, so it will not be affected by the presence of multiple IP subnets on the same bridged Ethernet, although detecting this situation automatically requires analysis of the router's interface and routing table.

Each bridge is queried with SNMP to retrieve its interface table and to build data structures needed to determine how it forwards data. To learn the MAC address of bridges and endpoints, the node running the algorithm pings each node. Sending that ping forces the querying node to learn the target's MAC address, which is recorded for use in analyzing the topology.

This initial phase of the operation weeds out members of both sets that are not responding properly. Bridges and hosts that appear to be down are dropped from the sets, and bridges that do not report a forwarding database are transferred to the endpoints set.

4.7.2 Learning

The next step in topology discovery is learning the forwarding database entries for each bridge. This is accomplished by walking the forwarding database table to download all of the entries. SNMP's GETNEXT command is issued to retrieve the first entry from the database. Repeated GETNEXT commands, indexed from the previous response, are used to traverse all entries in the database.

While the forwarding databases are being walked, the querying host is also pinging all members of the bridge and endpoint sets. This ensures that, minimally, all machines have entries in the FDBs of the bridges on the path between the querying host and the target machine.

Some bridges preserve expired entries or entries for themselves in the database. The self-referencing entries are removed. The expired entries may be used as a source of additional information. However, in the event that a machine has been moved and the bridge has not learned its new location, but other bridges have, this expired entry will generate conflicts when mapping the network. For the experiments described here, no expired entries were used—more than enough information was available through the current entries.

While many bridges arrange their forwarding database in ascending order, others provide it in an unsorted order—presumably the ordering of a hashtable. This has the unfortunate side effect that it is impossible to reliably walk the entire database. Because the entries in the database change continuously, walking the unsorted list may skip over entries or find loops. Sometimes only a few entries are received. Other times, the code to walk the database finds loops and fetches each entry many times. Currently, this difficulty is solved by aborting when a loop is followed too many times and making queries for each MAC address known in the bridge and endpoint sets, but not found while walking the table. This code could also be triggered if the walk completes, but the database was not ordered, and thus possibly hashed and shortened, but this has not been done to this point.

After the bridges' FDBs have been walked, an option for extra completeness is to go over each retrieved FDB and send queries to the bridge asking explicitly about any MAC belonging to a bridge or endpoint that is in the sets but not present in the database. Experience has shown that this option generally will not produce significantly more information. However, during the development of this algorithm, bridges have been found that do not support sequential walks of their database or that do not update the information available through the walk.[3] In such cases, other methods are necessary to obtain sufficient information from the bridges.

4.7.3 Deriving the topology

Once the databases have been loaded, the remainder of topology discovery is done internally. The first step of this procedure is to remove any unused bridges. A bridge that has been attached to the network but to which no host has been attached cannot be placed in the network by the mapping rules. Although it can be placed using entries for its own address, in the case where out-of-band connections are being used, this information may not be useful. Therefore, these bridges are dropped.

The algorithm performs the topology discovery by a traversal from the root bridge. The initial challenge is the selection of the best root. A heuristic is used that is designed to minimize the chances that a mapping between a pair of bridges may not meet the minimum knowledge requirement. The heuristic is therefore chosen to favor a bridge with entries distributed across many ports, rather than just one. Specifically, the bridge with the most FDB entries not mapping to the port with the most entries is selected. This heuristic was chosen because most bridges in the topology will have the majority of their entries mapped to their root port and because, due to various aspects of the learning phase, the root bridge may not have as many total entries as the leaf bridges. By excluding the entries from the most-used port, the algorithm tries to find a bridge with a large number of machines distributed across all its ports, thus meeting the minimum knowledge requirement.

Once the root has been selected, the remaining bridges are mapped to the port of the root bridge on which they appear. This algorithm forms the core of the topology discovery code. The only difference between the initial call of this algorithm and subsequent calls is that on the first call no bridge will have its root port selected, whereas on almost all subsequent calls that port will have been determined. The mapping algorithm will be described in detail below.

After mapping each bridge to one of the ports of the root bridge, the algorithm begins a simple traversal of the topology. For each port of the root bridge, the algorithm must determine which of the bridges connected to that port is the next hop, which is directly connected to that port. The algorithm attempts to map each child bridge as the next hop. A candidate is selected, and the algorithm maps the other bridges assigned to the root's port to their connections on the candidate. The correct next hop will have no other bridges connected to its root port. If any other bridges appear connected to the candidate's root port, then the candidate is not the next hop because there is a bridge in between the root bridge and the candidate. (Note that shared segments complicate the algorithm slightly and are discussed below.)

After determining the topology between the bridges, the endpoints are assigned to the topology. This is primarily a simple matter of observing the port on which the endpoint's MAC address appears that is not attached to another bridge. However, in cases where there

[3]This problem was believed to be caused by a bug in the bridge's OS and was solved by upgrading the bridge's software.

is insufficient information, the topology may need to be traversed to determine the final location of the host. This algorithm is discussed below.

4.7.4 Mapping algorithm

The core component of the topology discovery algorithm is the mapping algorithm. The mapping code allows for the explicit bridge-to-bridge forwarding information to be used or ignored. Normally it is useful, but if bridges are connected via out-of-band ports, it cannot be used. First, the procedure used to map nodes to ports using intersections will be described. Then the variations in the method for use of the explicit forwarding information will be described.

The basic approach to solving this problem is to calculate $T_a^x \cap T_b^y$ for each possible x and y. If the root port of b has been determined, the calculation requires only iteration over x. When a null intersection is found, that port assignment is returned as the correct match. If no match is found, an error is returned. If the minimum knowledge requiremet is met, this error should never happen, but in practice it can happen when an incorrect root is being checked and the bridge in question is very far away in the topology and not on the route between any pair of machines currently communicating with each other. In this situation returning an error is appropriate, because it will be passed up to a higher level where a new next root will be chosen.

The first special case is where the minimum knowledge requirement is not met. In this case, there are two possibilities. More information can be acquired, or the best can be done with what information is available. One method of acquiring more information would be to do the direct queries, if they hadn't been done before, in an attempt to complete the forwarding information. Currently, this approach is not taken automatically. The approach that is taken is to merely return that the child bridge is off of the root port of the root bridge. This has the effect that, if these two bridges are selected for the next root, a virtual switch will be inserted, connecting both of them to the previous root's port. While not necessarily the correct topology, this has the virtue of providing a solution that is consistent with all of the (inadequate) information that is available from the bridges.

A second precaution taken is that if the child's root port has not yet been determined, no requirement is made that it be found. Therefore, if the port of the root bridge on which the child is found is uniquely determined, but not vice versa, the mapping will still continue. At some point, either the child will be mapped off of a bridge for which it meets the minimum knowledge requirement, or an error will be generated.

Finally, the mapping routines also support a "named-only" mode, where only MACs belonging to endpoints with known names, i.e. E, are considered. In this case, MACs not belonging to E are removed from consideration when checking intersections. This option is used automatically if no other match can be found. The presence of out-of-band connections and broadcast addresses is believed to account for the need for this requirement

4.7.5 Virtual switches

In Section 4.5 I described two situations where a direct connection could not be established between two bridges in an otherwise complete network. The principal challenge in these situations is the creation of a shared Ethernet segment, rather than the modern point-to-point connections. These situations are:

1. A hub is used to connect two bridges with other hosts or bridges. Because hubs do not participate in the bridging algorithm, this creates a shared network segment between the bridges.

2. A bridge exists that the algorithm either was not informed above, or to which SNMP access is denied. Because SNMP security generally consists of a simple list of allowed or denied IP addresses, this situation can easily occur.

The direct connection theorem required a special case to handle these possibilities. However, using the simple connection theorem, no special case is required. This is because the theorem is used to map bridges to the port to which they are connected, rather than finding direct connections. In the course of the traversal, at each step the algorithm selects the next bridge in the tree. In the ideal case, a next hop will be found and will have no other bridges along its root port, therefore it is directly connected to the previous root. A virtual switch corresponding to one of the above conditions occurs when no single bridge is found to be the next hop. Instead, all bridges map at least one other bridge along their root ports, without causing any conflicts that would indicate that the previous root choice was incorrect.

To determine which bridges are connected directly to the virtual switch and which are their children, simple take the intersection of the nodes mapped to all of the bridges' root ports. This intersection will reveal the set of bridges that all bridges on this branch of the tree believe are on the link they use to connect to the previous root, and thus the set of bridges that must be connected to the virtual switch. The remaining bridges will exist only on the mapping belonging to one of the bridges on the shared segment, because they were mapped to the root port of the other bridges involved in the virtual switch, but excluded by the intersection.

The elegance of this solution is one of the appealing aspects of this technique. Rather than requiring a special case, virtual switches are naturally determined using the base algorithm.

4.7.6 Endpoint mapping

The final stage of the topology discovery algorithm is to map the endpoints to their locations on the topology. In most cases, this is very simple. The bridge to which the node is attached almost always had a current entry for each host attached to it in its forwarding database during the learning phase. If an endpoint's address is found on a bridge on a port to which no other bridge is attached, i.e. that port is a leaf in the topology, then that is the port to which the endpoint is attached. This handles almost all cases of endpoint mapping.

In a few cases, however, either the bridge happened to not return an entry for an endpoint directly connected to it when queried, or the endpoint is located on a shared segment between bridges. In this case, the graph is traversed from the root, looking for an entry for that endpoint. The goal of this algorithm is to prune away the parts of the bridge topology where the endpoint cannot be. As soon as an FDB entry for that endpoint is found, all bridges off of other ports of that bridge are eliminated. A traversal is begun from the port where the forwarding entry was found. Whenever a new forwarding entry is found, the bridges off of other ports are again pruned (in most cases including the bridge followed to this new node). Eventually, the algorithm either restricts the location of the endpoint to a link between two bridges, in which case the endpoint exists on a shared segment between the two bridges and a virtual switch is added to facilitate this location, or the endpoint is restricted to a set of bridges. In many cases, the solution turns out to contain one bridge, which is the bridge to which the node is actually connected, but which did not report an entry when queried. Here, the node is attached to a virtual switch at the point where the last information was provided. However, the code also queries the bridge for a forwarding entry for that endpoint, and this specific query has almost always resolved the location of the endpoint to a specific port on the second attempt to determine it.

4.7.7 Cleaning up the topology

The final step in producing a usable topology is to propagate the location of all nodes to all bridges in the topology. This ensures that it is simple to follow paths across the topology, even when the FDB entries for that node weren't present. When finished, a complete representation of the topology is available, with the information needed for automatic processing by applications.

4.8 Results

The topology discovery algorithm has been implemented as stand-alone code and as a module for use with the Remos system. The code was tested by running it on the extremely large Ethernet operated by the CMU CS department. With almost 2000 hosts and almost 50 bridges, it is doubtlessly one of the larger bridged networks in the world.

By far, the most time-consuming portion of the code is downloading the forwarding databases, which is simply a matter of the time it takes for the SNMP implementations on the querying machine and bridges to complete the data exchange. For this network, the time for these queries was approximately 45 minutes. The actually topology was calculated in less than 5 minutes.

The topology discovery algorithm has been run on several smaller networks. Although a few small changes were needed to support various incompatibilities in the SNMP support of the Ethernet bridges, the fundamental algorithm has performed flawlessly in all of the environments in which it has been tested.

Figure 4.16 shows the bridge topology discovered in the CS Department network. This topology was verified by the network manager as correct to the extent possible. There is

119

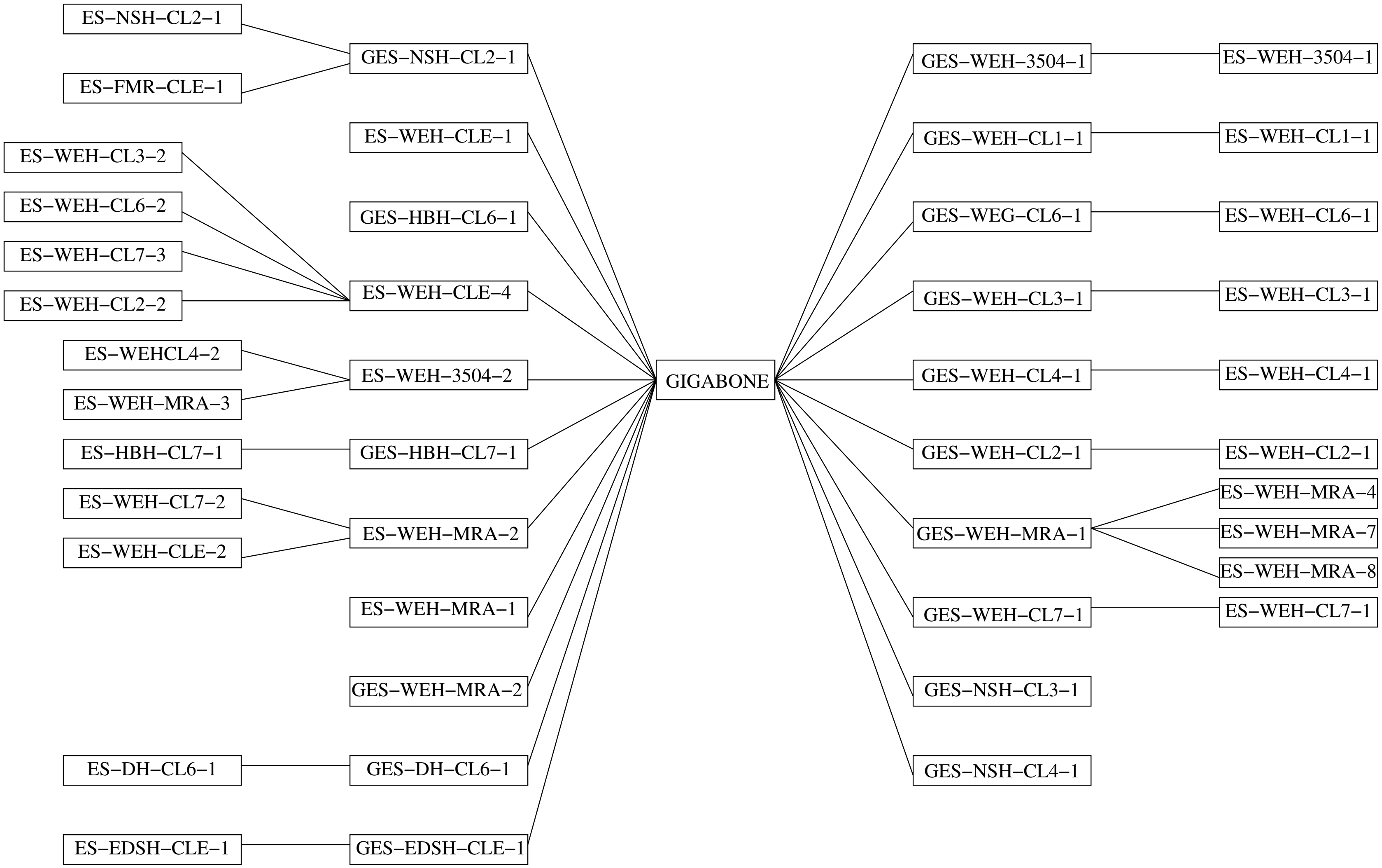


Figure 4.16: Topology of the CMU CS Department primary bridged Ethernet on May 29, 2000. 1831 endpoints are located on this topology and omitted for clarity.

no actual record of where hosts are attached to the network, but we discovered no errors in the placement of the machines we verified.

4.9 Practical considerations

My research has demonstrated that SNMP, already supported by almost all of the current networking infrastructure, is sufficient for obtaining the information needed to determine topology and predict performance directly from the most commonly-used networks used today. Although it is not an ideal interface for this purpose, it allows the network-based approach to performance prediction to be explored and utilized on existing networks. Demonstrating the value of this approach by using it in real systems and applications should result in the development of more appropriate interfaces for network components. However, both administrative and technical considerations must be addressed to provide a better interface for performance prediction purposes.

The administrative complication is primarily accessibility. Typically, SNMP access is only allowed from machines on the local network, and it is usually impossible to make SNMP queries to network components on an ISP's network. Security and privacy are the two primary reasons for this. Security is actually a technical concern; because the designers of SNMP were unable to agree on a workable security protocol, there is little security in current implementations, therefore a minimal security level is achieved by restricting access to local hosts. ISP's are generally concerned about privacy, not wishing to divulge information about the congestion levels of their services. Furthermore, because SNMP queries can be expensive, no one wants to open their network up to excessive load or even denial-of-service attacks with SNMP. We are currently pursuing combining network-based data with benchmark-based data to provide predictions in environments where direct network queries are only available for portions of the network.

Although RFCs describe the behavior of SNMP implementations, the standards and their implementations have not resulted in consistent interfaces between different manufacturers. For instance, the forwarding databases in Ethernet bridges are particularly troublesome. Some allow queries to be made for the forwarding port of a specific address. Other implementations are designed only for traversal, requiring the same query to be reformulated as a query for the subsequent entry from the numerically preceding address. Furthermore, some bridges remove the forwarding database if queries are made to it too rapidly, apparently as a security measure.

Additional networking technologies make topology discovery more complex. For instance, it is possible to run separate IP subnets on the same Ethernet. Although my algorithm and implementation correctly determine the topology in this situation, Remos is not currently able to recognize this configuration and provide accurate information to the application. Additionally, many network administrators are configuring their networks using VLAN technology. Although the same principals apply to discovering topology within VLANs, additional code is required to determine where the VLANs are sharing the same physical network connections. Much of the needed information appears to be available through standard MIBs, but I have not implemented this support in my software. Additional networking technologies will doubtlessly provide further challenges to topology discovery.

One of the motivations for this work is the lack of standardization between the various bridge vendors on support for topology discovery. While both Cisco and Intel, two major bridge vendors, support topology discovery, they use incompatible proprietary techniques for doing so. In September 2000, the IETF adopted the Physical Topology MIB as a standard MIB [14]. While an encouraging development, the earlier components of the RFC that specified a protocol to be used for topology discovery were removed from this RFC, and the RFC does not impose an Internet standard, merely reserves this portion of the MIB space. These shortcomings do not indicate a promising situation for the quick arrival of automatic topology discovery as a portable standard in commodity components.

4.10 Conclusions

The most important advantage of providing low-level network information to applications is the inclusion of the network's topology. Extraction of routing information from IP routed networks, ATM, and Myrinet is simple because the information is directly available. However, topology discovery on bridged Ethernet is quite challenging due to the transparent nature of the bridging algorithm. The bridging algorithm is one of the factors that has made Ethernet the most common LAN network, and applications must receive information about Ethernet topology to properly adapt to these networks.

The simple connection theorem presented here allows the topology of a bridged Ethernet to be determined even in the presence of incomplete knowledge. In particular, any bridge that has nodes detected off of two ports will be placed in the network topology. Because bridges are only used when multiple machines are to be connected to the network, this implies that nearly all Ethernet topologies will be detectable using these algorithms.

The significance of the simple connection theorem for bridge topology discovery is simply that it allows discovery to be performed where algorithms based on the direct connection theorem cannot be used. In the CS department network evaluated here, almost 2000 machines may be active at any given time. It is impossible to send pings from all machines, and the machines are rarely all connected for the continuous 45 minutes that would be required to collect complete forwarding databases from all bridges, if that would even be possible in a network this large. The simple connection theorem enables topology discovery because it can derive the topology with only a small amount of information from each part of the network.

The impact of topology information on parallel performance prediction is very important. Using topology, it is possible to detect bottleneck links without running an application. This is an important benefit, because bottlenecks induced by poorly planned topology are one of the largest differences between well-planned and poorly-planned distributed systems. Even without the network utilization techniques described in other chapters, topology prediction allows applications to detect the quality of the network they are using. The ability to detect and adapt to the difference between a Beowulf-class system and a collection of machines spread across different LANs is key in developing software to support network-nieve users on complex distributed systems.

Another possible use for network topology is planning benchmarks for network performance prediction. Although I have focused on techniques other than benchmarks for performance prediction, having accurate topology information allows benchmarks to be planned for maximum usefulness and minimal invasiveness. Scaling arguments aside, a combination of topology information, network-based utilization measurements, and selective benchmarks to measure the performance of congested portions of the network may provide the most accurate and useful predictions of network performance for applications.

Chapter 5

Conclusions

5.1 Summary

Access to accurate network information is essential to the performance of adaptive distributed applications. Most existing systems providing network information do so utilizing active benchmarks, or probes, in which data is sent across the network to determine the performance a future application will receive.

A number of projects have demonstrated that active benchmarking provides sufficient information for many applications, but there are still a number of shortcomings.

Topology I have presented several applications and common operations where the topology available through low-level network information can be used to enhance applications' performance running in distributed environments. The common operations include collective communication, cluster selection, and network performance metrics. Each of these examples are used for a wide variety of applications that ultimately benefit from access to this low-level information. Any application that plans its own communication pattern, selects clusters for multi-phased applications, or performs other similar activities can make direct use of topology information. Without this information, it is impossible to determine how bandwidth will be shared when multiple messages are sent simultaneously across the network, a key performance issue for many parallel applications.

Scalability Because only pairwise information is recorded by benchmarking, these approaches scale poorly if measurements are to be taken between all possible pairs of machines, requiring sacrifices to be made in either completeness or frequency of measurement. These costs can be reduced by collecting the data in a hierarchical fashion, but even a small LAN may contain too many machines to collect information about effectively. This restriction prevents benchmarking from being used in all environments. Without this range of functionality, benchmarking will never be able to support parallel applications with the same portability provided by current programming interfaces.

Invasiveness The third drawback to using application-level measurements is their invasiveness. They naturally consume the resource they are trying to measure. This becomes a significant problem when combined with the scalability problems already discussed.

The shortcomings of the application-based and benchmark-based measurement approaches motivate using low-level network information to meet the network-awareness requirements of the applications. The low-level information provides detailed topology information unavailable through the high-level approaches. By also using the low-level measurements to predict applications' end-to-end performance, the whole range of network information required by applications can be obtained using a scalable and non-invasive technique.

Remos has been designed to provide applications with both the low-level and high-level network information they require to adapt to the network. Remos provides a uniform interface so that portable network-aware applications can be developed independently of any particular network architecture.

The challenges in defining the Remos interface are network heterogeneity, diversity in traffic requirements, variability of the information, and resource sharing in the network. The Remos API is the result of an effort to present the network at as high a level of abstraction as possible, while maintaining the important low-level information needed by many applications. Remos supports both high-level flow-based queries and low-level topology queries. This combination allows Remos to meet the needs of the great majority of applications, including application classes that cannot be addressed with the information provided only through benchmarks.

To provide the information provided through the Remos interface, SNMP is used to obtain measurements directly from the network components. I have demonstrated that it is possible to provide accurate predictions of application-level performance using this low-level information. The current technique would have to be modified to be deployed at sites scattered across the Internet, but the accuracy of the predictions has been verified using both synthetic and real traces representing typical traffic found on both wide-area and local-area networks. These results give great promise for future deployment of this technique across a variety of environments.

Using SNMP, it is also possible to discover the topology of today's commonly used networks. Extraction of routing information from IP routed networks, ATM, and Myrinet is simple because the information is directly available. Topology discovery on bridged Ethernet is more challenging. I have presented an algorithm that allows the topology of a bridged Ethernet to be determined even in the presence of incomplete knowledge. In particular, the provable minimum knowledge requirement of my algorithm guarantees that it will complete successfully on practically any bridged Ethernet.

5.2 Contributions

- I have identified a several application classes and application support tools that benefit from or require knowledge of the network's topology. These examples motivate my approach to provide low-level information to applications in an accessible form. In particular, the local gap model, even though it provides a very high-level characterization of the network, cannot be used without access to network topology information.

- I have demonstrated that it is possible to provide network topology information and network component level performance characteristics while maintaining sufficient abstraction to ensure portability. The Remos interface allows applications to obtain both end-to-end performance predictions and network topology information within the same application-layer interface.

- I have developed a formalism for expressing the differences between various options for providing predictions of application performance on different networks. This formalism is particularly useful for comparing the complexity of producing application performance predictions using different sources of network performance data.

- My implementation of network-based performance prediction demonstrates that this technique has similar accuracy compared to benchmark-based prediction using the same competing traffic. Considering the scalability and invasiveness benefits of the network-based technique, the similar accuracy makes this a promising technique for environments where low-level network-based information can be obtained.

- I have developed an algorithm that can be used to derive the topology of a bridged Ethernet network with incomplete knowledge. Because it is difficult to obtain complete forwarding databases from Ethernet bridges, an algorithm that deals well with incomplete knowledge is needed to discover the topology of large Ethernet networks. My algorithm has provable behavior when faced with Ethernet bridges that do not provide forwarding entries for every address in the network.

5.3 Future work

A number of challenging issues and questions remain or have developed out of my research. Foremost among these issues is performing further verification of the network-based prediction technique, which is needed to validate its performance in different environments with different types of competing traffic. The traffic used in the experiments described in Chapter 3 was modeled after traffic aggregated from a number of flows. Further experiments need to be performed with more varieties of competing traffic, including traffic typical for a LAN, with fewer adaptive TCP flows; with WAN traffic accurately modeling many adaptive flows; and with traffic presenting a variety of flows, such as mixes of non-adaptive video and adaptive connections. Using a wider range of competing traffic sources will provide a better understanding of the advantages and disadvantages of different prediction techniques, as well as provide data to support the types of traffic data that should be collected by network components.

One interesting question about competing traffic that is not well understood is how burstiness in network traffic effects applications' performance. It is generally considered "common knowledge" that bursty traffic on a network causes problems for applications with fine granularity, but is less problematic for coarse-grained applications with longer-term bandwidth requirements. However, it isn't clear that this is true. Even if only caused by a temporary burst, packet loss can have long-term effects on TCP's performance. I have frequently been asked what the "correct" sampling interval is for measuring network utilization. I generally answer that it is application dependent, but I don't believe anyone really knows what range of intervals would be useful for different applications.

The history-based approach used for predictions in Chapter 3 was practical for those experiments, but is impractical for large-scale deployment. Before expanding the network-based prediction of end-to-end performance into a system that can be implemented across the Internet, the question of using history, performance models, or some combination must be addressed. Analysis of the techniques in a number of different environments with several different types of applications will aid in continuing this development.

Another interesting question is how more detailed information from the network components could be utilized. Although active networking may never be supported to the point that any user can download code into a router, networking technology does appear to be moving towards allowing network managers to insert their own code on their network components. I hope to acquire networking devices with these capabilities and use them to obtain more detailed statistics on the traffic passing through the network. My hope is that more detailed information will allow more accurate predictions to be made about applications' performance.

Combining the topology derived using network-based techniques with benchmark-based techniques remains an intriguing possibility. Using accurate topology information should allow benchmarks to be planned for maximum usefulness and minimal invasiveness. Topology information can be used to determine where the most useful parts of the network to measure are located and for planning traffic to minimize the invasiveness of the technique. Network-based utilization measurements can be made to determine which links are available and which are congested, and the most useful benchmarks can be selected based on this information. Scaling arguments aside, a combination of topology information, network-based utilization measurements, and selective benchmarks to measure the performance of congested portions of the network may provide the most accurate and useful predictions of network performance for applications.

Finally, my research has brushed on the curious relationship between network-aware applications and application-aware networks. In my mind, there is really a continuum here. There are ways in which a network can adapt or support an application that an application simply cannot do. Similarly, there are ways in which an application can adapt to the network that the network can never accomplish. Personally, I believe that the application has more degrees of freedom and have focused on network-aware applications. However, I think an ideal solution is one in which both types of adaptation can be used simultaneously. What is needed to develop such a solution is to bring together knowledge of how the application can adapt and knowledge of how the network can adapt together to work out a solution. It seems unlikely that there is a single representation that will describe all the different ways either applications or networks have of adapting to each other. But it might be possible to develop a programming representation where an application can describe how it would react to a particular network configuration and vice-versa. Then, a solution could be worked out iteratively with a master scheduler. There are a number of significant issues here, including the programming representation, scheduling algorithm, and security concerns. But I believe that with an appropriate secure computing environment, a solution is possible to better unify these two approaches to adaptation.

5.4 Conclusion

I have demonstrated that access to low-level network components provides the information needed by distributed applications to adapt themselves to their network environment. The techniques to obtain information from the network devices work with today's networking technology. Using that information, both the high-level end-to-end performance predictions and the low-level network topology needs of applications can be met. Using low-level network information provides information to applications in a scalable and non-invasive manner. While the needs of many applications are met with other current technologies, the scalability, non-invasiveness, and topology obtained through the network-based technique allow other applications, more resources, and more users to be used with and benefit from the power of distributed computing. Although the network-based technique violates the end-to-end abstraction provided by most networking protocols, the advantages of using low-level information at a higher level are clear.

Bibliography

[1] A. Adams, J. Mahdavi, M. Mathis, and V. Paxson. Creating a scalable architecture for internet monitoring. In *Proceedings of the 8th Annual Internet Society Conference (INET'98)*, Geneva, Switzerland, July 1998.

[2] I. Ahmad, Y. Kwok, and M. Wu. Analysis, Evaluation, and Comparison of Algorithms for Scheduling Task Graphs on Parallel Processors. In *Proceedings of the Second International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 207–213, June 1996 1996.

[3] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. In *Proceedings of the 10th International Parallel Processing Symposium IPPS'96*, pages 218–224. IEEE, April 1996.

[4] ATM User-Network Interface Specification. Version 4.0, 1996. ATM Forum document.

[5] S. Bajaj et al. Improving simulation for network research. Technical Report 99-702b, USC Computer Science Department, September 1999.

[6] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.-T. Ho, S. Kipnis, and M. Snir. CCL: A portable and tunable collective communication library for scalable parallel computers. In *Proceedings of 8th International Parallel Processing Symposium*, pages 835–844. IEEE Comput. Soc. Press, 1994.

[7] H. Balakrishnan, M. Stemm, S. Seshan, and R. H. Katz. Analyzing stability in wide-area network performance. In *Proceedings of SIGMETRICS'97*, pages 2–12. ACM, 1997.

[8] H. Bao, J. Bielak, O. Ghattas, L. F. Kallivokas, D. R. O'Hallaron, J. R. Shewchuk, and J. Xu. Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers. *Computer Methods in Applied Mechanics and Engineering*, 152(1–2):85–102, January 1998.

[9] M. Barnett, S. Gupta, D. G. Payne, L. Shuler, R. van de Geijn, and J. Watts. Building a high-performance collective communication library. In *Proceedings of IEEE Scalable High Performance Computing*, pages 835–834. IEEE Comput. Soc. Press, 1994.

[10] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceeding of the 1995 International Conference on Parallel Processing*, 1995.

[11] F. Berman and R. Wolski. The AppLeS project: A status report. In *Proceedings of the 8th NEC Research Symposium*, Berlin, Germany, May 1997.

[12] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing'96*, November 1996.

[13] P. Bhatt, V. Prasanna, and C. Raghavendra. Adaptive communication algorithms for distributed heterogrneous systems. In *Seventh IEEE Symposium on High-Performance Distributed Computing*, Chicago, IL, July 1998.

[14] A. Bierman and K. Jones. Physical topology MIB. RFC2922, September 2000.

[15] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet—a gigabit-per-second local-area network. *IEEE-Micro*, 15(1):29–36, February 1995.

[16] J. Bolliger and T. Gross. A framework-based approach to the development of network-aware applications. *IEEE Transactions on Software Engineering*, 24(5):376–90, May 1998.

[17] J. Bolliger, T. Gross, and U. Hengartner. Bandwidth modelling for network-aware applications. In *Proceedings of Infocomm'99*, 1999.

[18] F. Bonomi and K. Fendick. The rate-based flow control framework for the available bit rate atm service. *IEEE Network Magazine*, 9(2):25–39, March/April 1995.

[19] G. E. P. Box, G. M. Jenkins, and G. Reinsel. *Time Series Analysis: Forecasting and Control*. Prentice Hall, 3rd edition, 1994.

[20] Y. Breitbart, M. Garofalakis, C. Martin, R. Rastogi, S. Seshadri, and A. Silberschatz. Topology discovery in heterogeneous IP networks. In *Proceedings of INFOCOM 2000*, March 2000.

[21] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, and S. S. M. Karplus. CHARMM: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4:187–217, 1983.

[22] R. Busby, M. Neilsen, and D. Andresen. Enhancing NWS for use in an SNMP managed internetwork. In *Proceedings of the 2000 International Parallel and Distributed Processing Symposium (IPDPS'00)*, May 2000.

[23] CAIDA. Skitter. http://www.caida.org/TOOLS/measurement/skitter/.

[24] R. L. Carter and M. E. Crovella. Measuring bottleneck link speed in packet-switched networks. *Performance Evaluation*, 27 and 28, October 1996. also appears as Boston University BU-CS-96-006.

[25] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2), January 1999. RFC 1905.

[26] P. Chandra, A. Fisher, C. Kosak, and P. Steenkiste. Experimental evaluation of ATM flow control schemes. In *IEEE INFOCOM'97*, pages 1326–1334, Kobe, Japan, April 1996. IEEE.

[27] K. M. Chandy. Concurrent Program Archetypes. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, pages 1–9, October 1994.

[28] N. Chrisochoides, N. Mansour, and G. Fox. A comparison of optimization heuristics for the data mapping problem. *Concurrency: Practice and Experience*, 9(5):319–343, May 1997.

[29] D. Clark, S. Shenker, and L. Zhang. Supporting real-time applications in an integrated services packet network: Architecture and mechanisms. In *Proceedings of the SIGCOMM '92 Symposium on Communications Architectures and Protocols*, pages 14–26, Baltimore, August 1992. ACM.

[30] P. E. Crandall and M. J. Quinn. A partitioning advisory system for networked data-parallel processing. *Concurrency: Practice and Experience*, 7(5):479–495, August 1995.

[31] G. Davis and B. Massingill. The Mesh-Spectral Archetype. Technical report, Caltech, 1995.

[32] E. Decker, P. Langille, A. Rijsinghani, and K. McCloghrie. Definitions of managed objects for bridges. RFC1493, July 1993.

[33] R. Dietz. Remote monitoring mib extensions for application performance metrics. IETF Internet-Draft, July 1999. Work in progress.

[34] P. A. Dinda. *Resource Signal Prediction and Its Application to Real-time Scheduling Advisors*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 2000. Available as Carnegie Mellon University Computer Science Department Technical Report CMU-CS-00-131.

[35] P. A. Dinda and D. R. O'Hallaron. An evaluation of linear models for host load prediction. Technical Report CMU-CS-TR-98-148, School of Computer Science, Carnegie Mellon University, November 1998. A version of this paper will appear in HPDC '99.

[36] P. A. Dinda and D. R. O'Hallaron. An evaluation of linear models for host load prediction. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, August 1999.

[37] P. A. Dinda and D. R. O'Hallaron. An extensible toolkit for resource prediction in distributed systems. Technical Report CMU-CS-99-138, School of Computer Science, Carnegie Mellon University, July 1999.

[38] P. A. Dinda and D. R. O'Hallaron. Realistic CPU workloads through host load trace playback. In *Proc. of 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR2000)*, May 2000. To appear.

[39] D. Eckhardt and P. Steenkiste. A Wireless MAC with Service Guarantees. In preparation, 1998.

[40] W. Eddy, M. Fitzgerald, C. Genovese, A. Mockus, and D. Noll. Functional image analysis software—computational olio. In A. Prat, editor, *Proceedings in Computational Statistics*, pages 39–49, Heidelberg, 1996.

[41] K. Efe. Heuristic Models of Task Assignment Scheduling in Distributed Systems. *Computer*, pages 50–56, June 1982.

[42] H. El-Rewini and T. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9(2):138–53, 1990.

[43] D. J. Evans and W. Butt. Load Balancing with Network Partitioning Using Host Groups. *Parallel Computing*, 20:325–345, 1994.

[44] S. Figueira and F. Berman. Modeling the effects of contention on the performance of heterogeneous applications. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing (HPDC 5)*, pages 392–401, Syracuse, NY, August 1996.

[45] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems and Principles*, pages 48–63, December 1999.

[46] M. P. I. Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994. http://www.mcs.anl.gov/mpi/index.html.

[47] I. Foster and K. Kesselman. Globus: A metacomputing infrastructure toolkit. *Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[48] B. Gaidioz, R. Wolski, and B. Tourancheau. Synchronizing network probes to avoid measurement intrusiveness with the network weather service. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC 9)*, pages 147–154, Pittsburgh, PA, August 2000.

[49] G. A. Geist and V. S. Sunderam. The PVM System: Supercomputer Level Concurrent Computation on a Heterogeneous Network of Workstations. In *Proceedings of the Sixth Distributed Memory Computing Conference*, pages 258–261. IEEE, April 1991.

[50] N. Goddard, G. Hood, J. Cohen, W. Eddy, C. Genovese, D. Noll, and L. Nystrom. Online analysis of functional MRI datasets on parallel platforms. *The Journal of Supercomputing*, 11:295–318, 1997.

[51] R. Govindan and H. Tangmunarunkit. Heuristics for internet map discovery. In *IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.

[52] A. S. Grimshaw, W. A. Wulf, and T. L. Team. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), January 1997.

[53] E. L. Hahne. Round-robin scheduling for max-min fairness in data networks. *IEEE Journal on Selected Areas in Communication*, 9(7), September 1991.

[54] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. In *1996 ACM SIGMETRICS*, pages 13–24, May 1996.

[55] C. L. Hoover, J. Hansen, P. Koopman, and S. Tamboli. The amaranth framework: Probabilistic, utility-based quality of service management for high-assurance computing. In *IEEE Fourth International High-Assurance Systems Engineering Symposium (HASE'99)*, pages 207–216, Los Alamitos, CA, November 1999.

[56] C. Huang, E. P. Kasten, and P. K. Mckinley. Design and implementation of multicast operations for ATM-based high performance computing. In *Proceedings of Supercomputing '94*, pages 164–73, November 1994.

[57] C. Huang and P. K. McKinley. Design and implementation of global reduction operations across ATM networks. In *Proceedings of 3rd IEEE International Symposium on High Performance Distributed Computing*, pages 43–50. IEEE Comput. Soc. Press, 1994.

[58] J. Inouye, S. Cen, C. Pu, and J. Walpole. System support for mobile multimedia applications. In *Proceedings of the 7th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 143–154, St. Louis, May 1997.

[59] J. M. Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, July 1981.

[60] R. Jain. Congestion control and traffic management in ATM networks: Recent advances and a survey. *Computer Networks and ISDN Systems*, February 1995.

[61] S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. On the placement of internet instrumentation. In *IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.

[62] J.G.Donnett, M. Starkey, and D.B.Skillicorn. Effective Algorithms for Partitioning Distributed Programs. In *Proceedings of the Seventh Annual International Phoenix Conference on Computers and Communications*, pages 363–368, 1988.

[63] D. S. Katz and T. Cwik. Large-scale, low-cost parallel computers applied to reflector antenna analsysi. Presentation at the IEEE AP-S Symposium/URSI Radio Science Meeting, Atlanta, June 1998.

[64] D. S. Katz and T. Cwik. *High Performance Cluster Computing: Programming and Applications*, volume 2, chapter Computational Electromagnetics. Prentice Hall, 1999.

[65] D. S. Katz, T. Cwik, B. H. Kwan, J. Z. Lou, P. L. Springer, T. L. Sterling, and P. Wang. An assesment of a beowulf system for a wide class of analysis and design software. *Advances in Engineering Software*, 26(3–6):451–461, July 1998.

[66] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–308, February 1970.

[67] T. Kielmann, H. E. Bal, and S. Gorlatch. Bandwidth-efficient collective communication for clustered wide area systems. In *Proceedings of the International Parallel and Distributed Processing Symposium 2000 (IPDPS 2000)*, pages 492–499, May 2000.

[68] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 131–140, May 1999.

[69] P. kok Keong Loh, W. J. Hsu, C. Wentong, and N. Sriskanthan. How Network Topology Affects Dynamic Load Balancing. *IEEE Parallel and Distributed Technology*, pages 25–35, Fall 1996.

[70] H. Kung, T. Blackwell, and A. Chapman. Credit update protocol for flow-controlled ATM networks: Statistical mulitplexing and adaptive credit allocation. In *Proceedings of the SIGCOMM '94 Symposium on Communications Architectures and Protocols*, pages 101–114. ACM, August 1994.

[71] M. Laubach and J. Halpern. Classical IP and ARP over ATM. RFC2225, April 1998.

[72] M. Litzkow and M. Livny. Experience with the Condor Distributed Batch System. In *Proceedings. IEEE Workshop on Experimental Distributed Systems*, pages 97–101, October 1990.

[73] B. Lowekamp and A. Beguelin. ECO: Efficient collective operations for communication on heterogeneous networks. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, pages 399–405. IEEE, April 1996.

[74] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource monitoring system for network-aware applications. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 189–196. IEEE, July 1998.

[75] B. Lowekamp, D. O'Hallaron, and T. Gross. Direct queries for discovering network resource properties in a distributed environment. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 38–46. IEEE Computer Society, August 1999.

[76] B. Massingill. Mesh computations. Technical report, Caltech, 1995. caltech web site.

[77] K. McCloghrie and M. Rose. Management information base for network management of TCP/IP-based internets: MIB-II. RFC1213, March 1991.

[78] P. K. McKinley and J. W. S. Liu. Multicast tree construction in bus-based networks. *Communications of the ACM*, 33(1):29–41, January 1990.

[79] P. K. McKinley, Y. Tsai, and D. F. Robinson. A survey of collective communication in wormhole-routed massively parallel computers. Technical Report MSU-CPS-94-35, Michigan State University, June 1994.

[80] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883, Oregon, November 1993. ACM/IEEE.

[81] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):365–404, July 1976.

[82] N. Miller and P. Steenkiste. Collecting network status information for network-aware applications. In *IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.

[83] P. Mitra, D. G. Payne, L. Shuler, R. van de Geijn, and J. Watts. Fast collective communication libraries, please. Technical Report TR-95-22, The University of Texas, June 1995.

[84] G. J. Narlikar and G. E. Blelloch. Pthreads for dynamic and irregular parallelism. In *Proceedings of Supercomputing '98*, pages 801–824. IEEE, 1998.

[85] National Laboratory for Network Analysis (NLANR). Passive monitoring and analysis via packet header traces. http://moat.nlanr.net/. National Science Foundation Cooperative Agreement No. ANI-9807479.

[86] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, 1997.

[87] K. Obraczka and G. Gheorghiu. The performance of a service for network-aware applications. In *Proceedings of the ACM Sigmetrics SPDT'98*, 1998.

[88] D. R. O'Hallaron and J. R. Shewchuk. Properties of a Family of Parallel Finite Element Situations. Technical Report CMU-CS-96-141, Carnegie Mellon University, School of Computer Science, 1996.

[89] V. Paxson. Fast, approximate synthesis of fractional gaussian noise for generating self-similar network traffic. Technical Report LBL-36750, Lawrence Berkeley National Laboratory, April 1995.

[90] V. Paxson and S. Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[91] V. Paxson and S. Floyd. Why we don't know how to simulate the internet. In *Proceedings of the 1997 Winter Simulation Conference*, pages 1037–44, 1997.

[92] R. Perlman. *Interconnections: Bridges and Routers*. Addison-Wesley, 1992.

[93] V. J. Ribeiro, R. H. Riedi, M. S. Crouse, and R. G. Baraniuk. Simulation of non-gaussian long-range-dependent traffic using wavelets. In *Proceedings of ACM SIGMETRICS '99*, pages 1–12. ACM, May 1999.

[94] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions in Computer Systems*, 2(4):277–288, November 1984.

[95] A. Sang and S. qi Li. A predictability analsysis of network traffic. In *IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.

[96] A. Scherer, H. Lu, T. Gross, and W. Zwaenepoel. Transparent adaptive parallelism on NOWs using OpenMP. In *Proceedings of the Seventh ACM Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.

[97] J. M. Schopf and F. Berman. Performance prediction in production environments. In *Proceedings of IPPS/SPDP '98*, pages 647–653, 1998.

[98] S. Seshan, M. Stemm, and R. H. Katz. SPAND: Shared passing network performance discovery. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 135–46, December 1997.

[99] G. Shao, F. Berman, and R. Wolski. Using effective network views to promote distributed application performance. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, 1999.

[100] S. Sharma, R. Ponnusamy, B. Moon, Y. Hwang, R. Das, and J. Saltz. Run-time and compile-time support for adaptive irregular problems. In *Proceedings of Supercomputing '94*, pages 97–106, Washington, DC, Nov 1994.

[101] R. Siamwalla, R. Sharma, and S. Keshav. Discovering internet topology. http://www.cs.cornell.edu/skeshav/papers/discovery.pdf, July 1998.

[102] B. Siegell and P. Steenkiste. Automatic selection of load balancing parameters using compile-time and run-time information. *Concurrency - Practice and Experience*, 9(3):275–317, 1996.

[103] K. Sorber, S. Bernard, and H. El-Rewini. Performance Evaluation of Task Scheduling Techniques in Distributed Systems. In *Proceedings of the Fifteenth Annual Conference of the Chilean Computer Science Society*, pages 426–35, November 1995.

[104] W. Stallings. *SNMP, SNMPv2, and RMON*. Addison-Wesley, 2nd edition, 1996.

[105] A. Stathopoulos, A. Ynnerman, and C. F. Fischer. A PVM implementation of the MCHF atomic structure package. *International Journal of Supercomputer Applications and High Performance Computing*, 1995.

[106] P. Steenkiste. Adaptation models for network-aware distributed computation. In *Proceedings of the Third International Workshop on Network-Based Parallel Computing, Communication, Architecture, and Applications (CANPC'99)*, pages 16–31. Springer-Verlag, January 1999.

[107] J. Subhlok, P. Lieu, and B. Lowekamp. Automatic node selection for high performance applications on networks. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.

[108] J. Subhlok, P. Steenkiste, J. Stichnoth, and P. Lieu. Airshed pollution modeling: A case study in application development in an HPF environment. In *12th International Parallel Processing Symposium*, Orlando, FL, April 1998.

[109] J. Subhlok and G. Vondran. Optimal latency–throughput tradeoffs for data parallel pipelines. In *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 62–71, Padua, Italy, June 1996.

[110] J. Subhlok and B. Yang. A new model for integrated nested task and data parallel programming. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, June 1997.

[111] H. Tangmunarunkit and P. Steenkiste. Network-aware distributed computing: A case study. In *Second Workshop on Runtime Systems for Parallel Programming (RTSPP)*, page Proceedings to be published by Springer, Orlando, March 1998. IEEE. Held in conjunction with IPPS '98.

[112] W. Theilmann and K. Rothermel. Dynamic distance maps of the internet. In *IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.

[113] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[114] J. Weissman. Metascheduling: A scheduling model for metacomputing systems. In *Seventh IEEE Symposium on High-Performance Distributed Computing*, Chicago, IL, July 1998.

[115] J. B. Weissman and X. Zhao. Scheduling parallel applications in distributed networks. *Cluster Computing*, 1(1):95–108, May 1998.

[116] R. Wolski. Dynamically forecasting network performance using the network weather service. Technical Report CS-96-494, UCSD, 1996.

[117] R. Wolski. Forecasting network performance to support dynamic scheduling using the network weather service. In *Proceedings of the 6th High Performance Distributed Computing Conference (HPDC)*, pages 316–25, August 1997.

[118] R. Wolski. Personal communication on experiences with NWS, August 2000.

[119] R. Wolski, N. Spring, and C. Peterson. Implementing a performance forecasting system for metacomputing: The network weather service. In *Supercomputing '97*, 1997.

[120] T. Yang and C. Fu. Heuristic Algorithms for Scheduling Iterative Task Computations on Distributed Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):608–622, June 1997.